

# DEBUGGING MEMORY PROBLEMS USING TOTALVIEW



FEBRUARY 2005

VERSION 6.7

Copyright © 1999–2005 by Etnus LLC. All rights reserved.

Copyright © 1998–1999 by Etnus, Inc.

Copyright © 1996–1998 by Dolphin Interconnect Solutions, Inc.

Copyright © 1993–1996 by BBN Systems and Technologies, a division of BBN Corporation.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of Etnus LLC. (Etnus).

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Etnus has prepared this manual for the exclusive use of its customers, personnel, and licensees. The information in this manual is subject to change without notice, and should not be construed as a commitment by Etnus. Etnus assumes no responsibility for any errors that appear in this document.

TotalView and Etnus are registered trademarks of Etnus LLC.

TotalView uses a modified version of the Microline widget library. Under the terms of its license, you are entitled to use these modifications. The source code is available at <http://www.etnus.com/Products/TotalView/developers>.

All other brand names are the trademarks of their respective holders.

# Contents

## 1 Debugging Memory Problems

Checking for Problems .....	2
Programs and Memory .....	2
Behind the Scenes .....	5
Your Program's Data .....	7
The Data Section .....	8
The Stack .....	8
The Heap .....	12
Finding Allocation Problems .....	12
Finding Deallocation Problems .....	13
realloc() Problems .....	13
Finding Memory Leaks .....	13
Using the Memory Debugger .....	15
Memory Debugger Overview .....	15
Enabling, Stopping, and Starting .....	17
Finding free() and realloc() Problems .....	17
Event and Error Notification .....	18
Types of Problems .....	19
Freeing Unallocated Space .....	19
Freeing Memory That Is Already Freed .....	20
Tracking realloc() Problems .....	21
Freeing the Wrong Address .....	21
Block Properties and Event Notification .....	21
Finding Memory Leaks .....	24
Using Watch Points .....	26
Fixing Dangling Pointer Problems .....	26
Dangling Pointers .....	27
Examining Memory .....	28
Filtering .....	31
Block Painting .....	31
Hoarding .....	32
Example 1: Finding a Multithreading Problem .....	33
Example 2: Finding Dangling Pointer References .....	33

<b>2</b>	<b>Using the Memory Debugger Window</b>	
	About the Memory Debugger .....	35
	Common Operations .....	37
	Rows and Columns .....	37
	Filtering .....	38
	Saving Views .....	41
	Configuration Page .....	44
	Leak Detection Page .....	52
	Heap Status Page .....	56
	Memory Usage Page .....	60
<b>3</b>	<b>Using the dheap Command</b>	
	dheap Example .....	63
	dheap.....	65
	Notification When free Problems Occur .....	74
	Showing Backtrace Information: dheap -backtrace: .....	75
	Memory Reuse: dheap -hoard .....	75
	Writing Heap Information: dheap -export .....	77
	Filtering Heap Information: dheap -filter .....	77
	Checking for Dangling Pointers: dheap -is_dangling: .....	78
	Detecting Leaks: dheap -leaks .....	79
	Block Painting: dheap -paint .....	79
	Deallocation Notification: dheap -tag_alloc .....	80
	TV_HEAP_ARGS .....	82
<b>4</b>	<b>Creating Programs for Memory Debugging</b>	
	Linking Your Application With the Agent .....	83
	Attaching to Programs .....	85
	Using the Memory Debugger .....	86
	MPICH .....	86
	IBM PE .....	86
	SGI MPI .....	87
	RMS MPI .....	88
	Installing tvheap_mr.a on AIX .....	88
	LIBPATH and Linking .....	89
	Using the TVHEAP_ARGS Variable .....	90

# Debugging Memory Problems

# 1

Any time you read about debugging, you read that 60 or 70% of all programming errors are memory-related. So, while these numbers may be wrong, let's assume that they are right. Now for the bad news: the reason that memory errors occur is that the programmer made an error. All memory errors are preventable.

Why are there so many memory errors? There are many answers. For example, programs are complicated. And, programmers make assumptions when they shouldn't. Is a library function allocating its own memory or should the program be allocating it? Once it is allocated, does your program manage the memory or does the library? Something creates a pointer to something and the memory is freed without any knowledge that something else is pointing to it. Or, and these are the most prevalent reason, there's a wide separation between lines of code or the time when old code and new code was written. And, of course, there's always insufficient and bad documentation.

Some problems can be irrelevant. If you forget to free the memory allocated for a small array, it doesn't mean much. And, it may even be more efficient not to free the memory. The operating system will free it for you when the program ends, so there are times when you don't want to bother. On the other hand, if you continually allocate memory without freeing it, your program may eventually crash because it can't get more memory.

## Checking for Problems

---

The TotalView Memory Debugger can help you locate many of your program's memory problems. For example, you can:

- Stop execution when **free()**, **realloc()**, and other heap API problems occur.

If your program tries to free memory that it can't or shouldn't free, the Memory Debugger can stop execution. This lets you identify the statement that caused the problem. For more information, see "*Finding free() and realloc() Problems*" on page 17.

- List leaks.

The Memory Debugger can display your program's leaks. (*Leaks* are memory blocks that are allocated, but which are no longer referenced.)

When your program allocates a memory block, the Memory Debugger creates a backtrace.



*A backtrace is a list of stack frames. The Memory Debugger creates and stores the list of stack frames that are associated with many different kinds of memory events.*

When it makes a list of your leaks, it includes this backtrace in the list. This lets you see the place where your program allocated the memory block. For more information, see "*Finding Memory Leaks*" on page 24.

- Paint allocated and deallocated blocks.

When your program's memory manager allocates or deallocates memory, the Memory Debugger can write a bit pattern into it. Writing this bit pattern is called *painting*.

When you see this bit pattern in a Variable or Expression List Window, you know that you are using memory before your program initializes it or after your program deallocates it. Depending upon the architecture, you might even be able to force an exception when your program accesses this memory. For more information, see "*Block Painting*" on page 31.

- Identify dangling pointers.

*A dangling pointer* is a pointer that points into deallocated memory. If the pointer being displayed in a Variable Window is dangling, TotalView adds information to the data element so that you know about the problem. For more information, see "*Dangling Pointers*" on page 27.

- Hold onto deallocated memory.

When trying to identify memory problems, holding onto memory after your program releases it can sometimes help locate problems by forcing a memory error to occur. Holding onto freed memory is called *hoarding*.

If you are also painting memory, you can know when your program is trying to access deallocated memory. For more information, see "*Hoarding*" on page 32.

## Programs and Memory

---

When you run a program, your operating system loads the program into memory and defines an address space in which the program can operate.

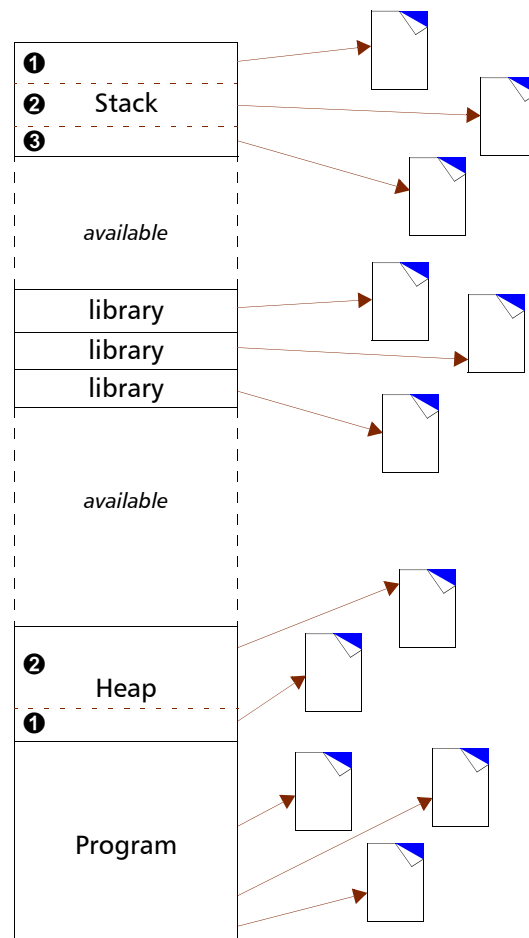
For example, if your program is executing in a 32-bit computer, the address space is approximately 4 gigabytes.



*Since the discussion in this chapter is pretty general, what you will be reading is almost true for many computer architectures, somewhat wrong for all, and perhaps completely wrong for the computer upon which you are debugging memory problems. For accurate information, you'll need to read information provided by your vendor.*

The operating system does not actually allocate the memory in this address space. Instead, operating systems memory map this space, which means that it maps the relationship between the theoretical address space your program could use and what it actually uses. Typically, operating systems divide memory into pages. When a program begins executing, the operating system creates a map that correlates the executing program with the pages that contain the program's information. The following figure shows regions of a program. The arrows point to the memory pages that contain the program.

Figure 1: Mapping Program Pages

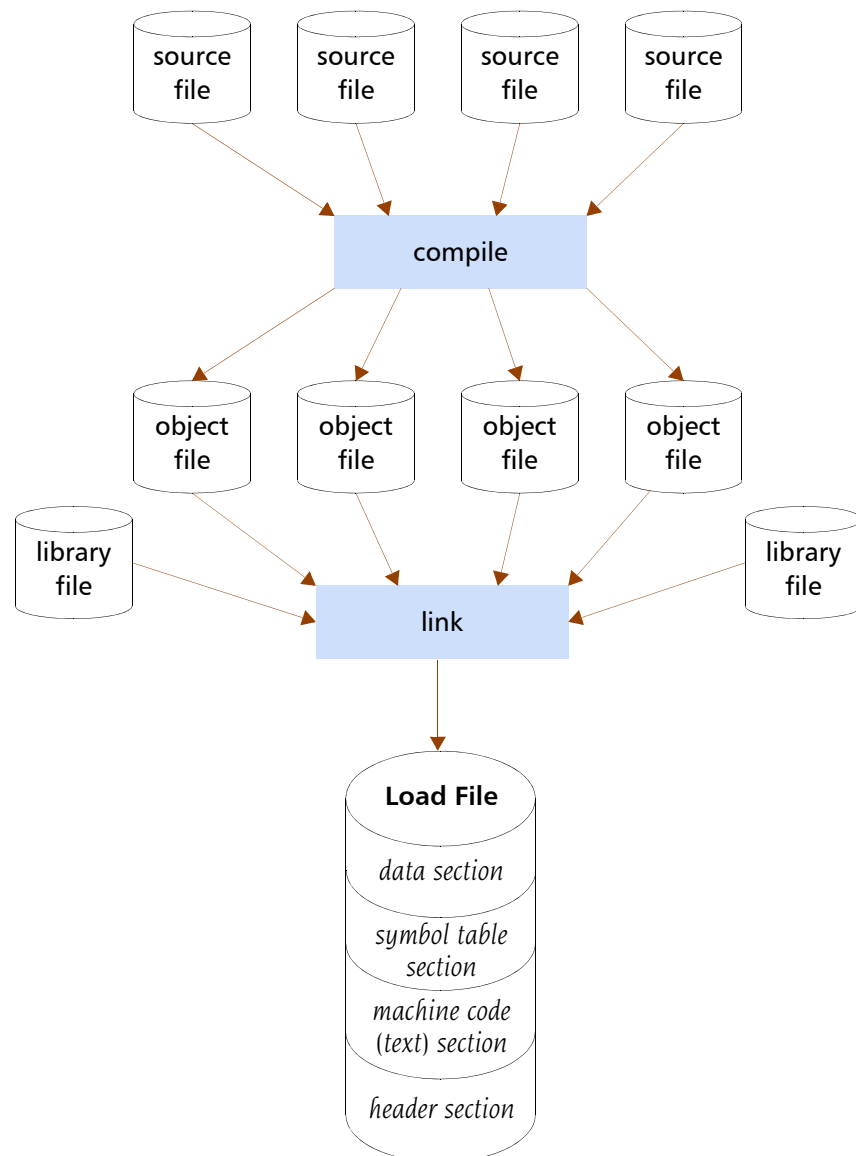


In this figure, the stack contains three stack frames, each mapped to its own page. Similarly, the heap shows two allocations, each of which is

mapped to its own page. (This isn't what really happens since a page can have many stack frames and many heap allocations. But doing this makes a nice picture.)

The program did not emerge fully-formed into this state. It had to be compiled, linked, and loaded. The following figure shows a program whose source code resides in four files. Running these files through a compiler creates object files. A linker then merges these object files and any external libraries needed into a load file. This load file is the executable program that is stored on your computer's file system.

Figure 2: Compiling Programs



When the linker creates the load file, it combines the information contained in each of the object files into one unit. Combining them is relatively straightforward. The load file shown at the bottom of this figure simplifies



this file's contents, since it always contains more sections and more information.

The contents of these sections are as follows:

- **Data section**—contains static variables and variables initialized outside of a function. The following is a small sample program:

```
int my_var1 = 10;
void main ()
{
    static int my_var2 = 1;
    int my_var3;
    my_var3 = my_var1 + my_var2;
    printf("here's what I've got: %i\n", my_var3);
}
```

The data section contains the **my\_var1** and **my\_var2** variables. The memory for the **my\_var3** variable is dynamically and automatically allocated within the stack by your program's runtime system.

- **Symbol table section**—contains addresses (usually offsets) to the locations of routines and variables.
- **Machine code section**—contains an intermediate binary representation of your program. (It is intermediate because addresses are not yet resolved.)
- **Header section**—contains information about the size and location of information in all other sections of the object file.

When the linker creates the load file from the object and library files, it interweaves these sections into one file. The linking operation creates something that your operating system can load into memory. Figure 3 on page 6 shows this process.

The Memory Debugger can provide information about these sections and the amount of memory your program is using. To obtain this information, select the **Tools > Memory Debugging** command and then select the **Memory Usage** tab and select Process View. (See Figure 4 on page 7.)

In this listing, the data and symbol table sections of the load file are combined into the **Data** column.

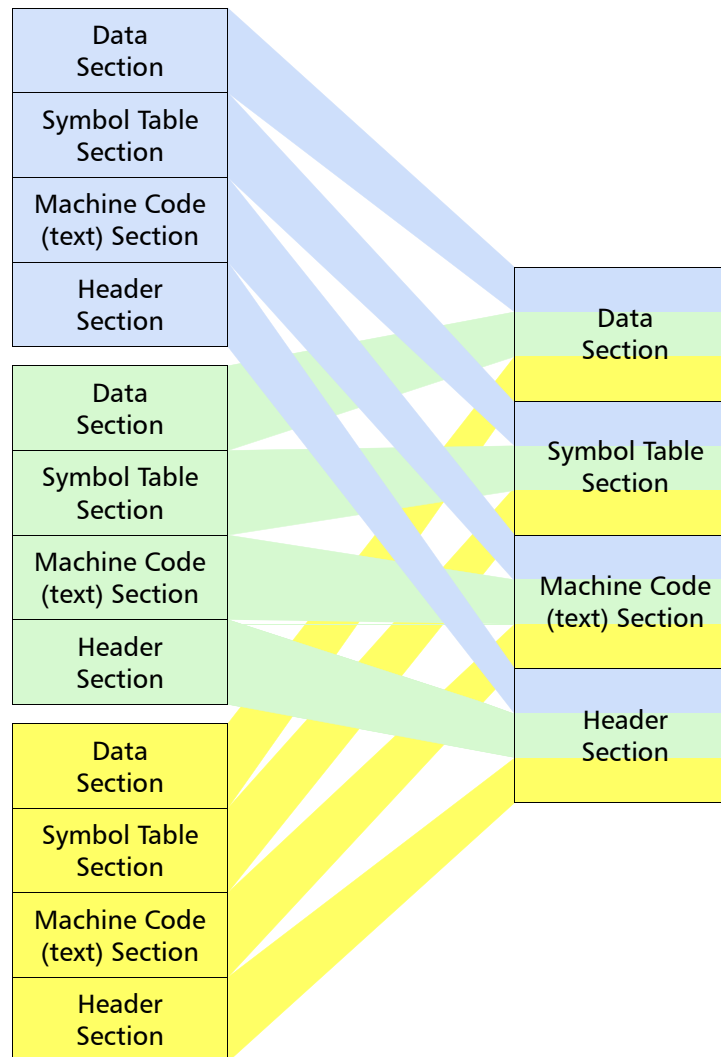
For information on this page, see "Memory Usage Page" on page 60.

## Behind the Scenes

The TotalView Memory Debugger intercepts calls made by your program to heap library functions that allocate and deallocate memory using the **malloc()** and **free()** functions and the **new** and **delete** operators. It also tracks related functions such, as **calloc()** and **realloc()**. The Memory Debugger uses a technique called interposition, in which an agent intercepts calls to functions.

You can use the Memory Debugger with any allocation and deallocation library that uses such functions as **malloc()** and **free()**. For example, the C++ **new** operator is almost always built on top of the **malloc()** function. If it is, the Memory Debugger can track it. Similarly, some Fortran implemen-

Figure 3: Linking a Program

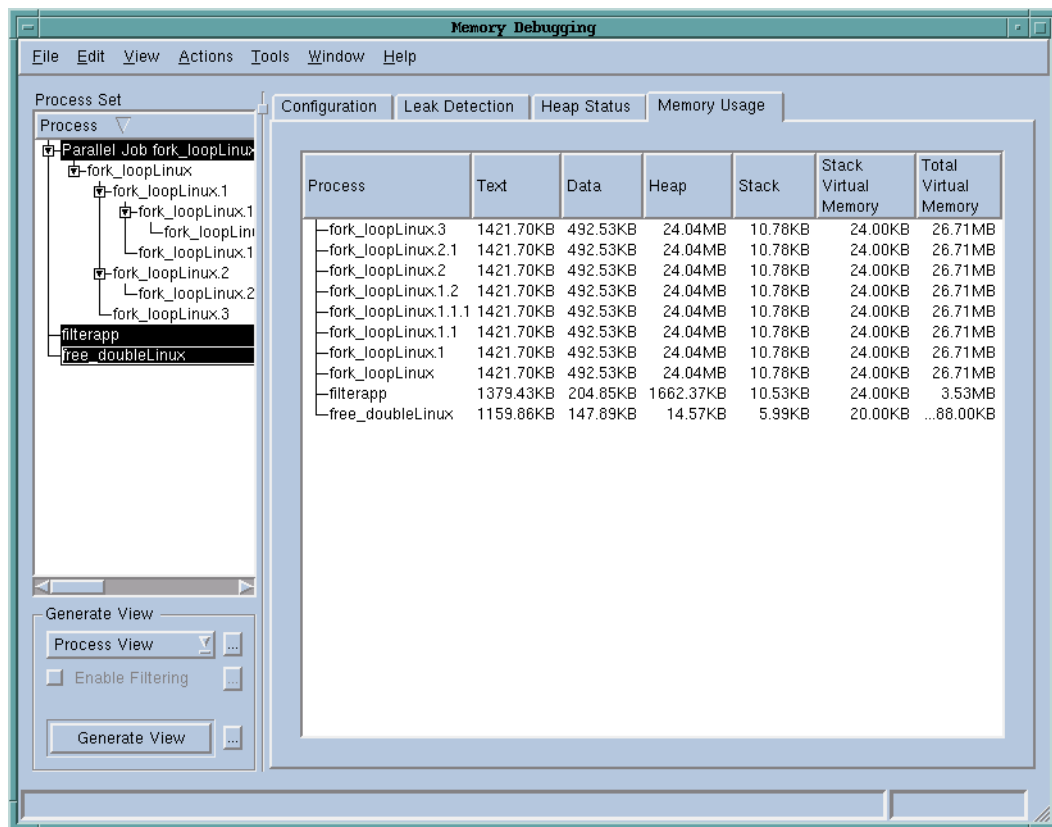


tations use the **malloc()** and **free()** functions to manage memory. In these cases, the Memory Debugger can track Fortran memory use.

You can interpose the agent in two ways:

- You can tell TotalView to preload the agent. *Preloading* means that the loader loads an object before the object listed in the application's loader table. When a routine references a symbol in another routine, the linker searches for the first definition of that symbol. Because the agent's routine is the first object in the table, its routine is invoked instead of the routine in the program's heap manager. On Linux, HP Tru64 Alpha, Sun, and SGI, TotalView sets an environment variable that contains the pathname of the agent's shared library in your local TotalView installation. For more information, see "Attaching to Programs" on page 85.
- If TotalView cannot preload the agent, you must explicitly link it into your program. For details, see "Creating Programs for Memory Debugging" on page 83.

Figure 4: Memory Usage Page: Process View



If your program attaches to an already running program, you must explicitly link this other program with the agent.

The agent uses operations defined in the dynamic linker's API to find the original definition of the routine. After the agent intercepts a call, it calls the original function. This means that you can use the Memory Debugger with most memory allocators. Figure 5 on page 8 shows how the agent interacts with your program and the heap library.

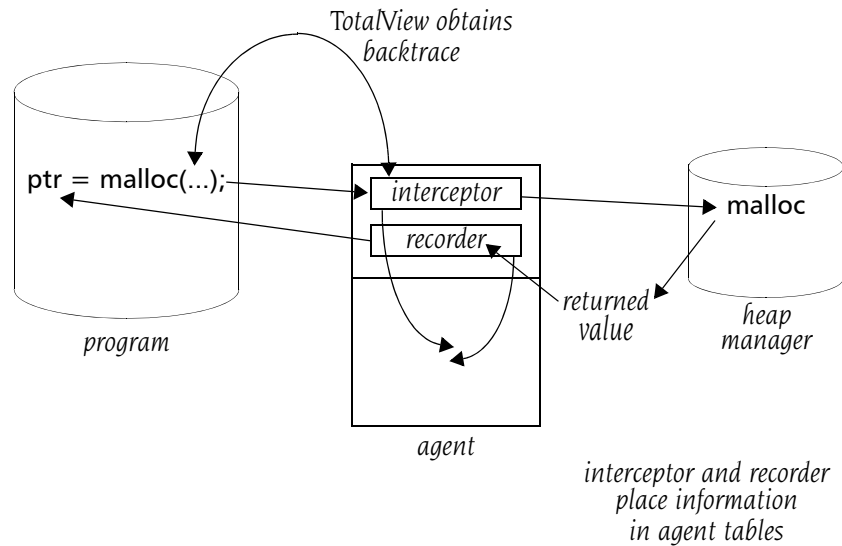
Because TotalView uses interposition, memory debugging can be considered non-invasive. That is, TotalView doesn't rewrite or augment your program's code, and you don't have to do anything in your program. Adding the agent does not change your program's behavior.

## Your Program's Data

Your program's variables resides in the following places:

- Data section
- Stack
- Heap

Figure 5: Interposition



## The Data Section

Memory in the data section is permanently allocated. Your program uses this section for storing static and global variables. The size of this section is fixed when the operating system loads the program and the variables within it exist for the entire time that your program is executing. Errors can occur if your program tries to manage this section's memory. For example, you cannot free memory allocated to variables in the data section. In general, errors are usually related to the programmer not understanding that the program can't manage data section memory.

## The Stack

Memory in the stack section is dynamically managed by your program's memory manager. Consequently, your program cannot allocate memory within the stack or deallocate memory within it.



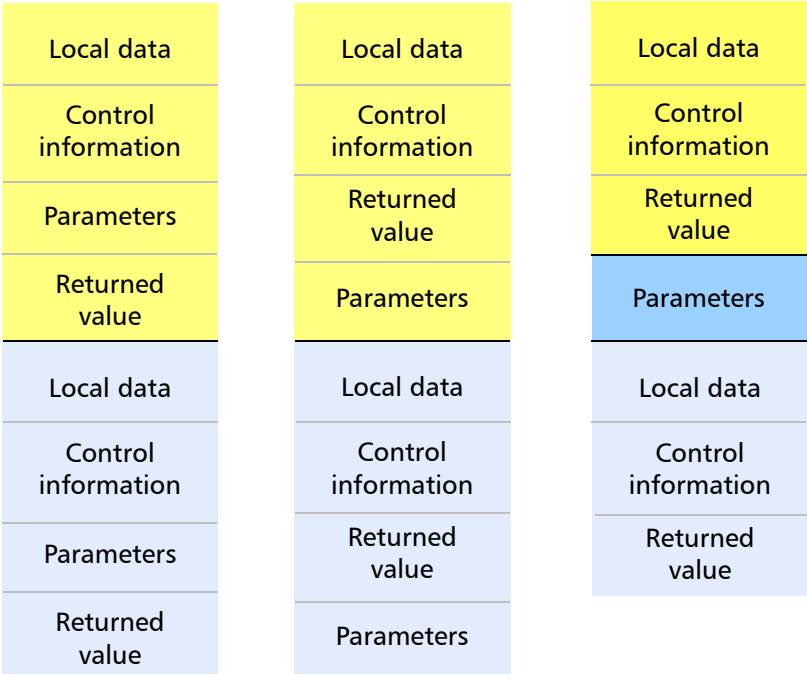
*"Deallocates means that your program is no longer using this memory. The next time your program calls a routine, the new stack frame overwrites the memory previously used by other routines. In almost all cases, deallocated memory, whether on the stack or the heap, just hangs around in its preallocation state until it gets reassigned."*

The stack differs from the data section in that the space is dynamically managed. What's in it one minute might not be there a moment later. Your program's runtime environment allocates memory for stack frames as your program calls routines and deallocates these frames when execution exits from it.

At a minimum, a stack frame contains lots of control information, data storage, and space for passed-in arguments (parameters) and the returned value. Figure 6 on page 9 shows three ways in which a compiler can arrange stack frame information:

In this figure, the left and center stack frames have different positions for the parameters and returned value. The stack frame on the right is a little

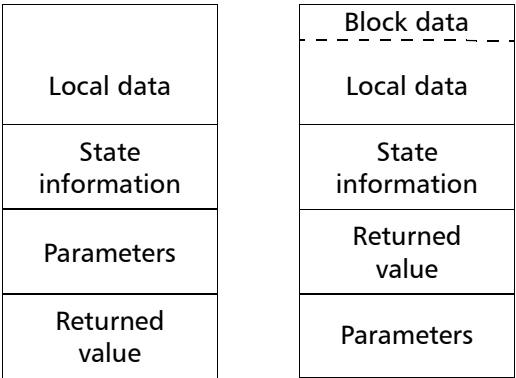
Figure 6: Placing Parameters



more complicated. In this version, the parameters are located within a stack memory area that doesn't belong to either stack frame.

If a stack frame contains local (sometimes called automatic) variables, where is this memory placed? If the routine has blocks in which memory is allocated, where on the stack is this memory for these additional variables placed? Although there are many variations, the following figure shows two of the more common ways to allocate memory:

Figure 7: Local Data in a Stack Frame

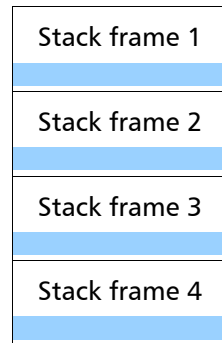


The blocks on the left shows a data block allocated within a stack frame on a system that ignores your routine's block structure. The compiler figures how much memory is needed, and then allocates enough memory for all of your routine's automatic variables. These kinds of systems are optimized to minimize the time necessary to allocate memory. Other systems dynamically allocate the memory required for a block as the block is entered, and

then deallocate it as execution leaves the block. (The blocks on the right show this.) These kinds of systems are optimized to minimize a routine's size.

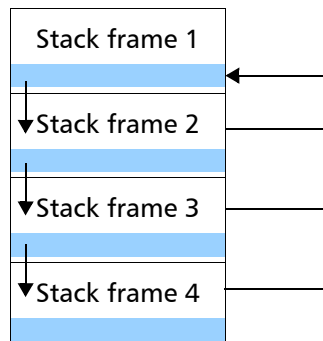
As your program executes routines, routines call other routines, placing additional routines on the stack. The following figure shows four stack frames. The shaded areas represents local data.

Figure 8: Four Stack Frames



What happens when a pointer to memory in a stack frame is passed to lower frames? This situation is shown in the following figure:

Figure 9: Passing Pointers



The arrows on the left represent the pointer passed down the stack. The lines and arrows on the right indicate the place to which the pointer is pointing. A pointer to memory in frame 1 is passed to frame 2, which passes the pointer to frame 3, and then to frame 4. In all frames, the pointer points to a memory location in frame 1. Stated in another way, the pointers in frames 2, 3, and 4 point to memory in another stack frame. This is considered the most efficient way for your program to pass data from one routine to another. Using the pointer, you can both access and alter the information that the pointer is pointing to.

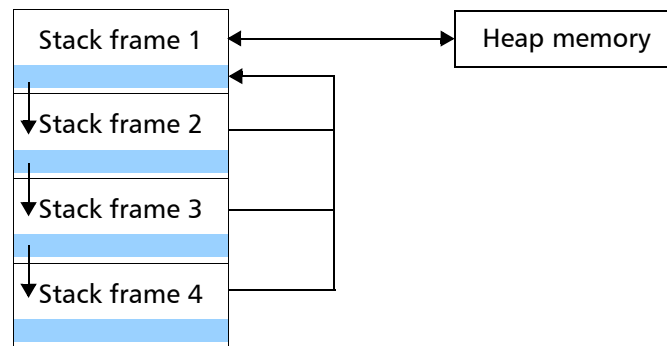


*Sometimes you read that data can be passed by-value (which means copying it) or by-reference (which means passing a pointer). This really isn't true. Something is always copied. "Pass-by-reference" means that instead of copying the data, the program copies a pointer to the data.*

Because the program's run-time system owns stack memory, you cannot free it. Instead, it gets freed when a frame is popped from the stack.

One of the reasons for memory problems is that you it may sometimes be unclear who owns a variable's memory. For example, in the following figure, the routine in frame 1 has allocated memory in the heap, and passes a pointer to that memory to other stack frames:

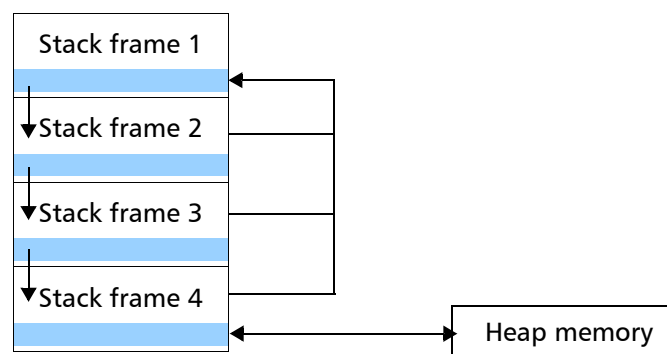
Figure 10: Allocating a Memory Block



If the routine executing in frame 4 frees this memory, all pointers to that memory are dangling; that is, they point to deallocated memory. If the program's memory manager reallocates this heap memory block, the data accessible by all the pointers is both invalid and wrong. Unfortunately, if the memory manager doesn't immediately reuse the block, the data accessed through the pointers is still correct. This is unfortunate, because there's no guarantee that the data is correct and there won't be any pattern to when the block becomes invalid. This means that when problems occur, they are intermittent, which makes them even harder to locate.

Another common problem is when you allocate memory and assign its location to an automatic variable. This is shown in Figure 11 on page 11.

Figure 11: Allocating a Block from a Stack Frame

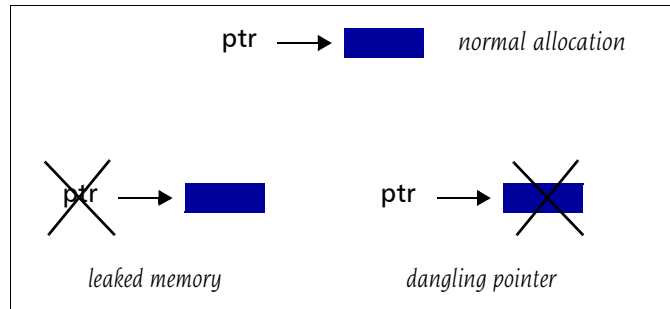


If frame 4 returns control to frame 3 without deallocating the heap memory it created, this memory is no longer accessible. That is, your program loses the ability to use this memory block. It has *leaked* this memory block.



If you have trouble remembering the difference between a leak and a dangling pointer, this may help. Before either problem occurs, memory is created on the heap and the address of this memory block is assigned to a pointer. A leak occurs when the pointer gets deleted, leaving a block with no reference. In contrast, a dangling pointer occurs when the memory block is deallocated, leaving a pointer that points to deallocated memory. Both are shown in the following figure.

Figure 12: Leaks and Dangling Pointers



The Memory Debugger Leak Detection Page shows all of your program's leaks. For information on detecting leaks, see "Finding Memory Leaks" on page 24.

## The Heap

The *heap* is an area of memory that your program uses when it wants to dynamically allocate space for data. While using the heap gives you a considerable amount of flexibility, you must manage this resource. You allocate and deallocate this space. In contrast, you do not allocate or deallocate memory in other areas.

Because allocation and deallocation are intimately linked with your program's algorithms and, in some cases, the way you use this memory is implicit rather than explicit, problems associated with the heap are the hardest to find.

### Finding Allocation Problems

Memory allocation problems are seldom due to allocation requests. Instead, they occur because your program either is using too much memory or is leaking it. Because an operating system's virtual memory space is large, allocation requests usually succeed. Nevertheless, you should always check the value returned from allocation requests such as `malloc()`, `calloc()`, and `realloc()`. Similarly, you should always check whether the C++ `new` operator returns a null pointer. (Newer C++ compilers throw a `bad_alloc` exception.) If your compiler supports the `new_handler` operator, you can throw your own exception.

You can tell the Memory Debugger to stop execution when your program encounter memory allocation problems. However, since these problems are rare, you might never come across one.



## Finding Deallocation Problems

The Memory Debugger can let you know when your program encounters a problem deallocating memory. Some of the problems it can identify are:

- **free not allocated:** An application calls the **free()** function using an address that is not in a block allocated in the heap.
- **realloc not allocated:** An application calls the **realloc()** function using an address that is not in a block allocated in the heap.
- **Address not at start of block:** A **free()** or **realloc()** function receives a heap address that is not at the start of a previously allocated block.

If a library routine use the memory manager and a problem occurs, the Memory Debugger still locates the problem. For example, the **strdup()** string library functions call the **malloc()** function to create memory for a duplicated string. Since the **strdup()** function is calling the **malloc()** function, the Memory Debugger can track this memory.

You can tell the Memory Debugger to stop execution just before your program misuses a heap API operation. This lets you see what the problem is before it actually occurs. (For more information, see "Behind the Scenes" on page 5.)



*Because execution stops before your program's heap manager deallocates memory, you can use the **Thread > Set PC** command to set the PC to a line after the free request. This means that you can continue debugging past a problem that might cause your program to crash.*

## realloc() Problems

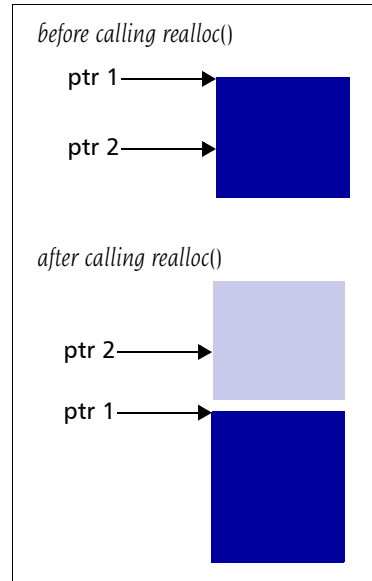
The **realloc()** function can create unanticipated problems. This function can either extend a current memory block, or create a new block and free the old. Although you can check to see which action occurred, you need to code defensively so that problems do not occur. Specifically, you must change every pointer pointing to the memory block to point to the new one. Also, if the pointer doesn't point to the beginning of the block, you need to take some corrective action.

In the following figure, two pointers are pointing to a block. After the **realloc()** function executes, **ptr1** points to the new block. However, **ptr2** still points to the original block, a block that was deallocated and returned to the heap manager. (See Figure 13 on page 14.)

## Finding Memory Leaks

Technically, there's no such thing as a memory leak. Memory doesn't leak, can't leak. With that said, a memory leak is a block of memory that a program allocates that is no longer referenced. For example, when your program allocates memory, it assigns the block's location to a pointer. A leak can occur if one of the following occurs:

- You assign a different value to that pointer.
- The pointer was a local variable and execution exited from the block.

Figure 13: *realloc()* Problem

If your program leaks a lot of memory, it can run out of memory. Even if it doesn't run out of memory, your program's memory footprint becomes larger. This increases the amount of paging that occurs as your program executes. Increased paging makes your program run slower.

Here are some of the circumstances in which memory leaks occur:

- **Orphaned ownership**—your program creates memory but does not preserve the address so that it can deallocate it at a later time.

The following example makes this (extremely) obvious:

```
char *str;
for( i = 1; i <= 10; i++ )
{
    str = (char *)malloc(10*i);
}
free( str );
```

Within the loop, your program allocates a block of memory and assigns its address to **str**. However, each loop iteration overwrites the address of the previously created block. Because the address of the previously allocated block is lost, its memory can never be made available to your program.

- **Concealed allocation**—the action of creating a memory block is separate from its use.

As an example, contrast the **strcpy()** and **strdup()** functions. Both do the same thing: they make a copy of a string. However, the **strdup()** function uses the **malloc()** function to create the memory it needs, while the **strcpy()** function uses a buffer that your program creates.

In general, you must understand what responsibilities you have for allocating and managing memory. For example, when your program receives a handle from a library, the handle allows you to identify a memory block allocated by the library. When you pass the handle back to the library, it knows what memory block contains the data you want to use or manipu-

late. There may be a considerable amount of memory associated with the handle, and deleting the handle without deallocating the memory associated with the handle leaks memory.

- **Changes in custody**—the routine creating a memory block is not the routine that frees it. (This is related to concealed allocation.)

For example, routine 2 asks routine 1 to create a memory block. At a later time, routine 2 passes a reference to this memory to routine 3. Which of these blocks is responsible for freeing the block?

This type of problem is more difficult than other types of problems in that it is not clear when the data is no longer needed. The only thing that seems to work consistently is reference counting. In other words, when routine 2 gets a memory block, it increments a counter. When it passes a pointer to routine 3, routine 3 also increments the counter. When routine 2 stops executing, it decrements the counter. If it is zero, the executing routine frees the memory. If it isn't zero, another routine frees it at another time.

- **Underwritten destructors**:—when a C++ object creates memory, it must ensure that its destructor frees it. No exceptions. This doesn't mean that a block of memory cannot be allocated and used as a general buffer. It just means that when an object is destroyed, it needs to completely clean up after itself.

For more information, see “*Finding free() and realloc() Problems*” on page 17.

## Using the Memory Debugger

Here is how you start the TotalView Memory Debugger:

- 1 Enable the Memory Debugger from within the Memory Debugger Window or the CLI. You must enable the Memory Debugger before execution begins.
- 2 Tell the Memory Debugger what operations to perform. These operations include hoarding, painting, and telling it to notify you when problems occur using the heap library. *Notification* means that the Memory Debugger stops a program's execution when problems using the heap API occur.

Whenever your program is stopped—for example, it is at a breakpoint or you halted it—you can tell the Memory Debugger to create a view that describes any program leaks or a report that describes currently allocated memory blocks.

### Memory Debugger Overview

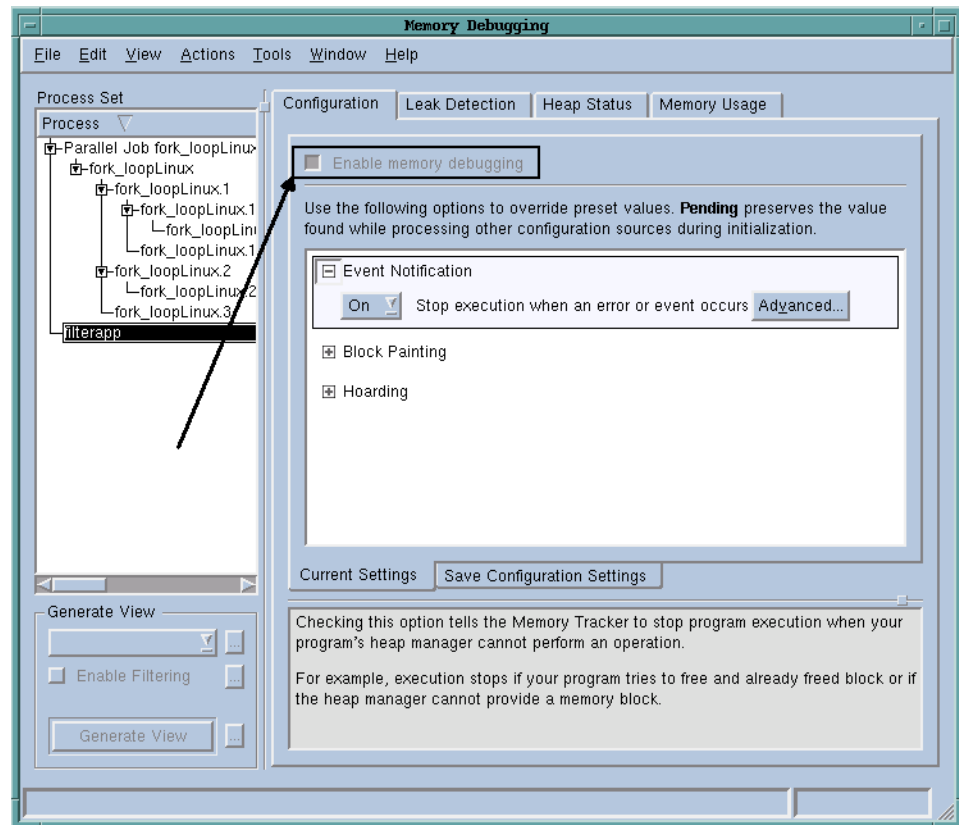
TotalView must be able to preload your program with the Memory Debugger agent. In many cases, it can do this automatically. However, you must manually link the agent if your application involves remote debugging. In addition, TotalView cannot preload the agent for applications that run on IBM RS/6000 platforms. For more information, see “*Creating Programs for Memory Debugging*” on page 83.

## Using the Memory Debugger

The following procedure describes how you begin using the Memory Debugger:

- 1 After you start TotalView but before you start executing your program, select the **Tools > Memory Debugging** command. The displayed window shows the Configuration Page.

Figure 14: Configuration Page



- 2 Before configuring the Memory Debugger, select one or more of the processes shown in the **Process Set** area on the left.
- 3 If the **Enable memory debugging** check box isn't checked, you need to select it. If you have explicitly linked your program with the agent, TotalView automatically checks it for you.
- 4 Start your program and run it to a breakpoint.

Before your program begins execution, you will need to set other options in the Configuration Page:

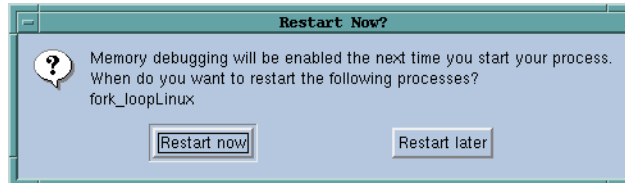
- **Memory Event Notification**—tells the Memory Debugger to stop execution and notify you if a heap event such as a deallocation or a problem occurs. (See "Event and Error Notification" on page 18 for more information.)
- **Memory Block Painting**—tells the Memory Debugger to paint allocated and deallocated memory and the pattern that the Memory Debugger uses when it paints this memory. For more information, see "Finding free() and realloc() Problems" on page 17 and "Block Painting" on page 47.

## Enabling, Stopping, and Starting

- **Memory Hoarding**—tells the Memory Debugger to hoard deallocated memory blocks, the size of the hoard, and the number of blocks that the hoard can contain. For more information, see “Hoarding” on page 49.

If your program is executing, you cannot enable or disable the Memory Debugger. If you try, TotalView displays its **Restart Now?** Dialog Box:

Figure 15: Restart Now Dialog Box



Selecting **Restart now** tells TotalView to kill your program, enable or disable the Memory Debugger, and then restart your program. If you select **Restart later**, your program continues executing. After you restart your program, the Memory Debugger will do what you asked it to.

If you turn on notification and all you want to do is stop TotalView from notifying you about heap problems, Remove the check mark from the Configuration Page’s **Stop execution when error or event occurs** check box. While the Memory Debugger continues to track memory events, it no longer stops execution if a problem occurs. Of course, your operating system might terminate execution when an error occurs. However, your program might continue executing. For example, many systems ignore a **free()** request that tries to free memory that your program already freed.

Telling the Memory Debugger not to notify you when a problem occurs is useful. For example, suppose you are calling functions in a shared library, and you aren’t interested in or can’t debug this code and the library has heap problems. Turning off notification lets you execute past this code. Do this by setting a breakpoint at a location after the library function executes. When execution stops, enable notification.

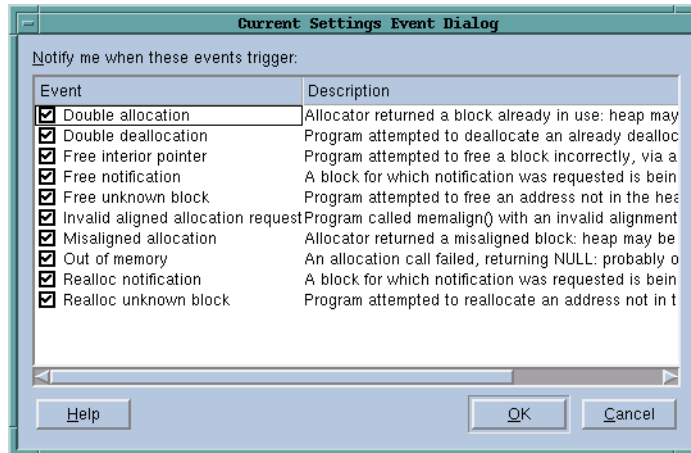
## Finding free() and realloc() Problems

The Memory Debugger detects problems that occur when you allocate, reallocate, and free heap memory. This memory is usually allocated by the **malloc()**, **calloc()**, and **realloc()** functions, and deallocated by the **free()** and **realloc()** functions. In C++, the Memory Debugger tracks the **new** and **delete** operators. If your Fortran libraries use the heap API, the Memory Debugger tracks your Fortran program’s dynamic memory use. Some Fortran systems use the heap API for assumed-shape, automatic, and allocatable arrays. See your system’s **man** pages and other documentation for more information.

### Event and Error Notification

After you enable memory debugging and turn on notification, TotalView stops execution if it detects a notifiable event such as a free problem. There are a number of events that can cause the Memory Debugger to stop execution. If you select the **Advanced** button within the Memory Debugger's Configuration Page, the Memory Debugger displays a dialog box that lets you specify which of memory events will stop execution.

Figure 16: Advanced



When execution stops, the PC is at an internal TotalView breakpoint. As the following figure shows, the lines above the breakpoint have information about what to do next.

Figure 17: TotalView Internal Memory Breakpoint

```

31 | /*
32 |  * TotalView has stopped your process because it detected
33 |  * a heap event. For more information:
34 |  *
35 |  * - (GUI) See the message in the Memory Block Event dialog box
36 |  * - (CLI) Type "dheap -status"
37 |  */
38 |
39 | TV_HEAP_event = *event;
40 | } /* TV_HEAP_notify_breakpoint_here () */
41 |

```

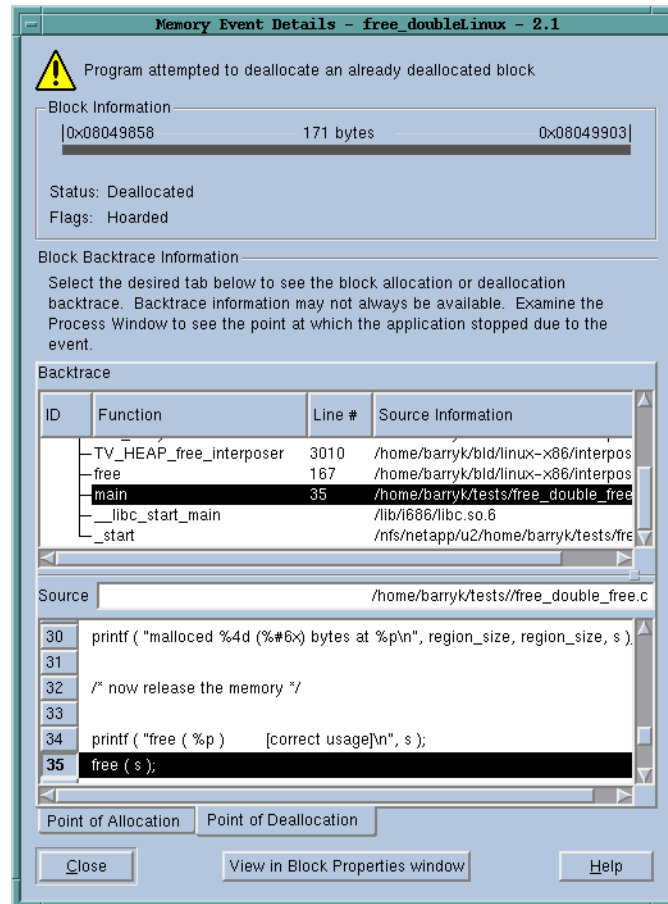
TotalView also displays its **Memory Event Details** Window (see Figure 18 on page 19):

This window has four areas, as follows:

- The top line tells you what type of error or event occurred.
- The **Block Information** area gives the memory location of the block and its status.
- The third area contains the function backtrace if the error or event is related to a block allocated on the heap. The Memory Debugger retains information about the backtrace that existed when the memory block was allocated and the backtrace when it was deallocated. You can tell the Memory Debugger which it should display by selecting either the **Point of Allocation** or **Point of Deallocation** tab.

If a memory error occurred, the deallocation backtrace is often the same as the backtrace being shown in the Process Window's Source Pane. If the

Figure 18: Memory Error Block Window



memory error occurs after your program deallocated this memory, the backtraces are different.

- The bottom area shows you where the allocation or deallocation occurred in your program.



*In some cases, the Memory Debugger does not display an allocation backtrace. For example, if you try to free memory allocated on the stack or in a data section, there's no backtrace because your program did not allocate the memory.*

If you need to redisplay the Memory Block Window after you dismiss it, select the **Tools > Memory Event Details** command.

## Types of Problems

This section presents some trivial programs that illustrate some of the **free()** and **realloc()** problems that the Memory Debugger detects. The errors shown in these programs are obvious. Errors in your program are, of course, more subtle.

### Freeing Unallocated Space

The following section contains programs that free space that they cannot deallocate.

### Freeing Stack Memory

The following program allocates stack memory for the `stack_addr` variable. Because the memory was allocated on the stack, the program cannot deallocate it.

```
int main (int argc, char *argv[])
{
    void *stack_addr = &stack_addr;
    /* Error: freeing a stack address */
    free(stack_addr);
    return 0;
}
```

### Freeing bss Data

The bss section contains uninitialized data. That is, variables in this section have a name and a size but they do not have a value. Specifically, these variables are your program's uninitialized static and global variables. Because they are contained in a data section, your program cannot free their memory.

The following program tries to free a variable in this section:

```
/* Not initialized; should be in bss */
static int bss_var;

int main (int argc, char *argv[])
{
    void *addr = (void *) (&bss_var);
    /* Error: address in bss section */
    free(addr);
    return 0;
}
```

### Freeing Data Section Memory

If your program initializes static and global variables, it places them in your executable's data section. Your program cannot free this memory.

The following program tries to free a variable in this section:

```
/* Initialized; should be in data section */
static int data_var = 9;

int main (int argc, char *argv[])
{
    void *addr = (void *) (&data_var);
    /* Error: address in data section */
    free(addr);
    return 0;
}
```

### Freeing Memory That Is Already Freed

The following program allocates some memory, then releases it twice. On some operating systems, your program can SEGV on the second free request.

```
int main ()
{
    void *s;
    /* Get some memory */
    s = malloc(sizeof(int)*200);
    /* Now release the memory */
    free(s);
}
```



```

    /* Error: Release it again */
    free(s);
    return 0;
}

```

### Tracking realloc() Problems

The following program passes a misaligned address to the **realloc()** function.

```

int main (int argc, char *argv[])
{
    char *s, *misaligned_s, *realloc_s;

    /* Get some memory */
    s = malloc(sizeof(int)*64);
    /* Reallocate memory using a misaligned address */
    misaligned_s = s + 8;
    realloc_s = realloc(misaligned_s, sizeof(int)*256);
    return 0;
}

```

In a similar fashion, TotalView detects **realloc()** problems caused by passing addresses to memory sections whose memory cannot be released. For example, TotalView detects problems if you try to do the following:

- Reallocate stack memory.
- Reallocate memory in the data section.
- Reallocate memory in the bss section.

### Freeing the Wrong Address

TotalView can detect when a program tries to free a block that does not correspond to the start of a block allocated using the **malloc()** function. The following program illustrates this problem:

```

int main (int argc, char *argv[])
{
    char *s, *misaligned_s;

    /* Get some memory */
    s = malloc(sizeof(int)*64);
    /* Release memory using a misaligned address */
    misaligned_s = s + 8;
    free(misaligned_s);
    free(s);
    return 0;
}

```

## Block Properties and Event Notification

When an error occurs, such as those discussed in “Types of Problems” on page 19, the Memory Debugger stops program execution. (The Memory Debugger can also stop execution when your program deallocates or reallocates a memory block.) For example, if your program tries to free memory already freed, the Memory Debugger stops execution. However, you won’t know where and when this memory was first freed. This section describes a procedure that tells the Memory debugger to give you this information.

## Finding free() and realloc() Problems

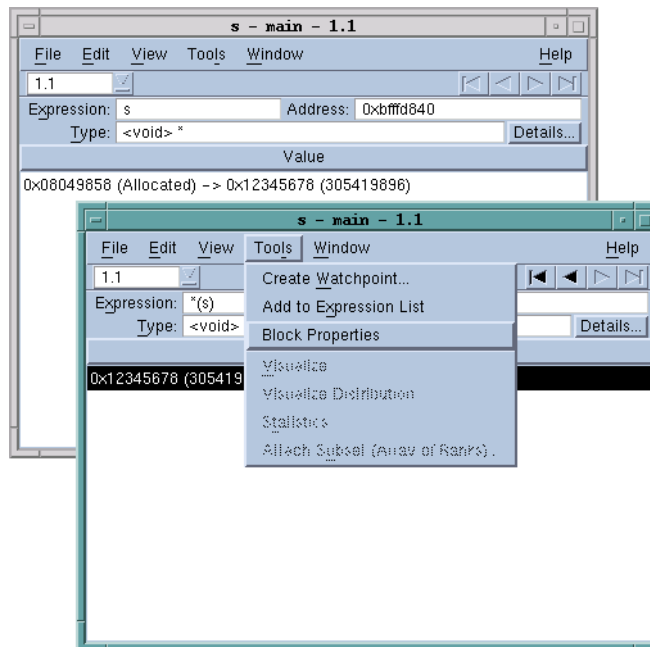
Here's a trivial program that contains a double free error:

```
01 int main ()
02 {
03     void *s;
04     /* Get some memory */
05     s = malloc(sizeof(int)*200);
06     /* Now release the memory */
07     free(s);
08     /* Error: Release it again */
09     free(s);
10     return 0;
11 }
```

Here's the procedure:

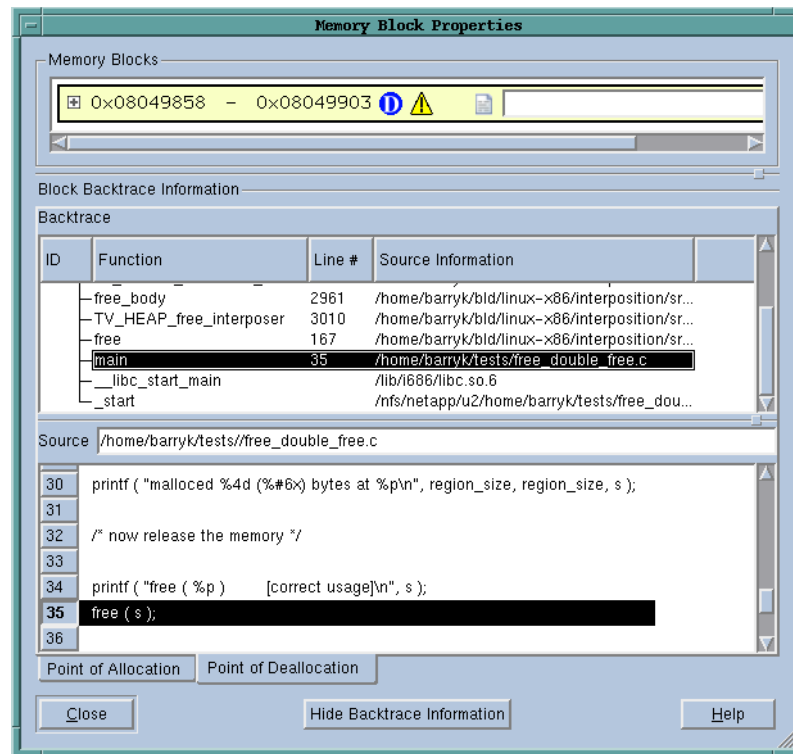
- 1 Display the Memory Debugging Window by selecting the **Tools > Memory Debugging** command.
- 2 After enabling memory debugging, check **On** within the **Event Notification** area.
- 3 Select line 07 in the Process Window and press the **Run To** button in the toolbar.
- 4 Dive on variable **s**. In the displayed Variable Window, dive on the pointer value.

Figure 19: Variable Window for Pointer *s*



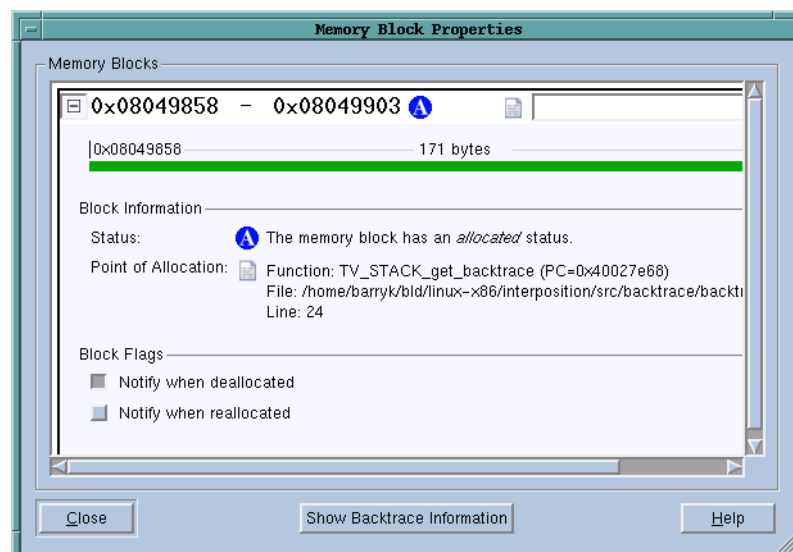
- 5 After selecting on the pointer's value, select the **Tools > Block Properties** command. The Memory Debugger displays it's **Block Properties** Window.

Figure 20: Memory Block Properties Window



The control that tells the Memory Debugger to notify you when the block is freed is within the top Memory Blocks area. You can either expand the area and press the + symbol or you can press the **Hide Backtrace Information** button at the bottom of this window. If you press this button, you'll see the following window:

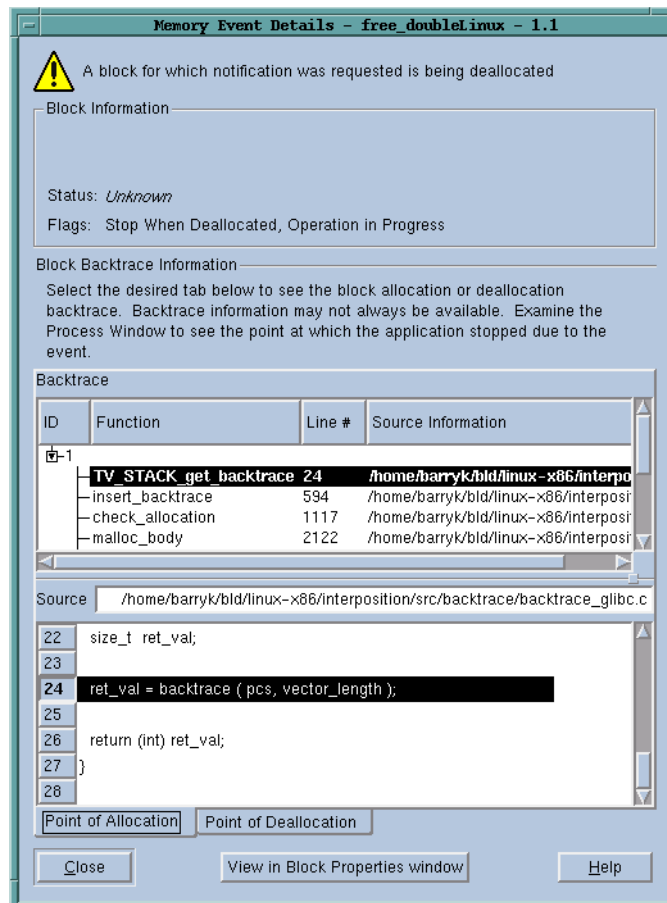
Figure 21: Memory Block Properties Window



After selecting the **Notify when deallocated** check box, close the window. Selecting this button tells the Memory Debugger to monitor this memory block such that when your program frees it, it should stop execution and let you know that this just occurred.

- 6 Select the **Go** button from the toolbar. After line 07 executes, the Memory Debugger stops execution and displays the **Memory Event Details** Window.

Figure 22: Memory Block Properties Window



Using procedures similar to this, you can track any deallocations that might be interesting.

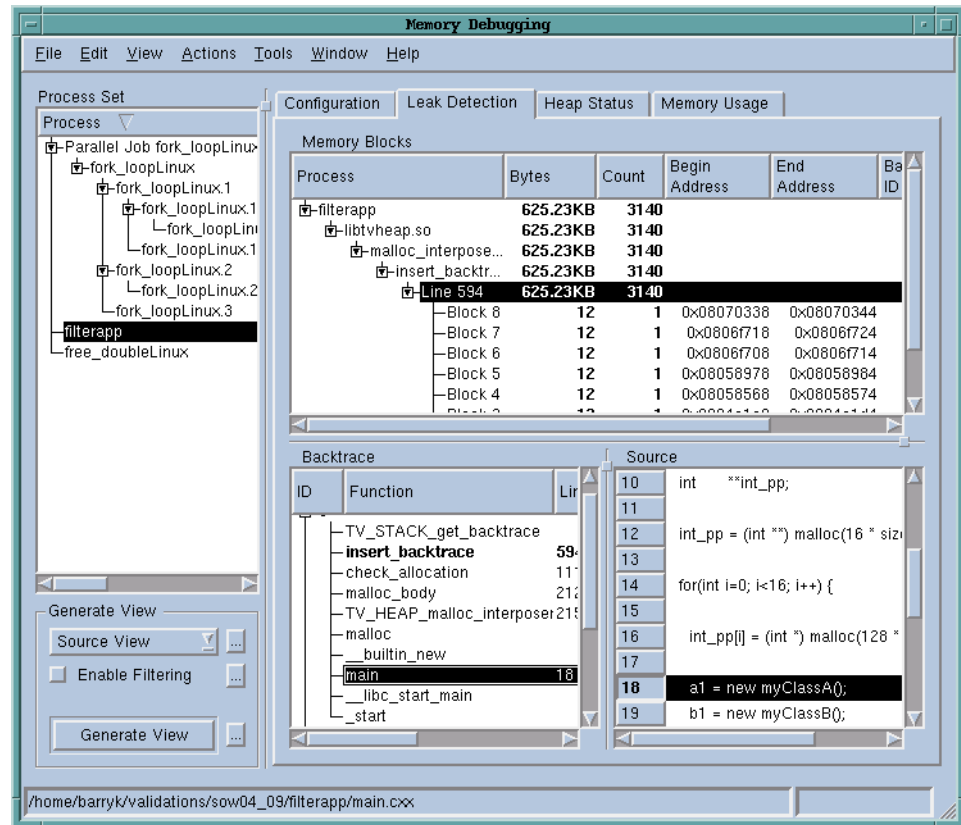
## Finding Memory Leaks

The TotalView Memory Debugger can locate your program's memory leaks and display information about them.

- 1 Before execution begins, enable the Memory Debugger. (See "Enabling, Stopping, and Starting" on page 17.)
- 2 Run the program and then halt it where you want to look at memory problems. Allow your program to run for a while before stopping execution to give it enough time to create leaks.

- From the **Memory Debugger Window** (invoked using the **Tools > Memory Debugging** command), select the **Leak Detection** tab. (See Figure 23 on page 25.)

Figure 23: Leak Detection  
Page: Source View



- Select one or more processes in the **Process Set** area.
- Select a view within the **Generate View** area and click the **Generate View** button. For example, you might select **Source View**.
- Examine the list. After you select a leak in the top part of the window, the bottom of the window shows a backtrace of the place where the memory was allocated. After you select a stack frame in the backtrace, TotalView displays the statement where the block was created.

The backtrace that the Memory Debugger displays is the backtrace that existed when your program made the heap allocation request. *It is not the current backtrace.*

The line number displayed in the Memory Debugger Source Pane is the same line number that TotalView displays in the Process Window Source Pane. If you go to that location, you can begin devising a strategy for fixing the problem. Sometimes you get lucky and the fix is obvious. In most cases, it isn't clear what was (or should be) the last statement to access a memory block. Even if you figure it out, it's extremely difficult to determine if the place you located is really the last place your program needs this data. At this point, it just takes patience to follow your program's logic.

Many users like to generate a view that contains all leaks for the entire program. Do this by setting a breakpoint on your program's **exit** statement. After your program stops executing, generate a Leak Detection View.

### Using Watch Points

For many types of memory problems, identifying where the problem occurred is just the first step. Your next step is to look for the solution. TotalView and the Memory Debugger can help. For example, here's a procedure that lets you identify when your program writes to a memory block:

- 1 Using the backtrace in the Leak Detection Page, identify where your program allocated the memory.
- 2 Go to the Process Window and set a breakpoint after that line.
- 3 Restart your program and run it to that breakpoint.
- 4 Dive on the pointer and, if it is not automatically dereferenced, dive on the pointer in the Variable Window.
- 5 Select the **Tools > Watchpoint** command and set a watchpoint.
- 6 Select **Go**.

Your program stops executing when the value contained at this memory location changes. If there are a number of statements in your program that write into this memory location, you might need to select **Go** a number of times. Eventually, you will know when the last time your program changes a value. Watchpoints do not, unfortunately, get triggered when your program reads data.

## Fixing Dangling Pointer Problems

---

Fixing dangling pointer problems is usually more difficult than fixing other memory problems. First of all, you only become aware of them when you realize that the information your program is manipulating isn't what it is supposed to be. Even more troubling, these problems can be intermittent, happening only when your program's heap manager reuses a memory block. For example, if nothing else is running on your computer, the block might never be reused. If there are a large number of jobs running, a deallocated block could be reused quickly.

After you identify that you have a dangling pointer problem, you have two problems to solve. The first is to determine where your program freed the memory block. The second is to determine where it *should* free this memory. Memory Debugger tools that can help you are:

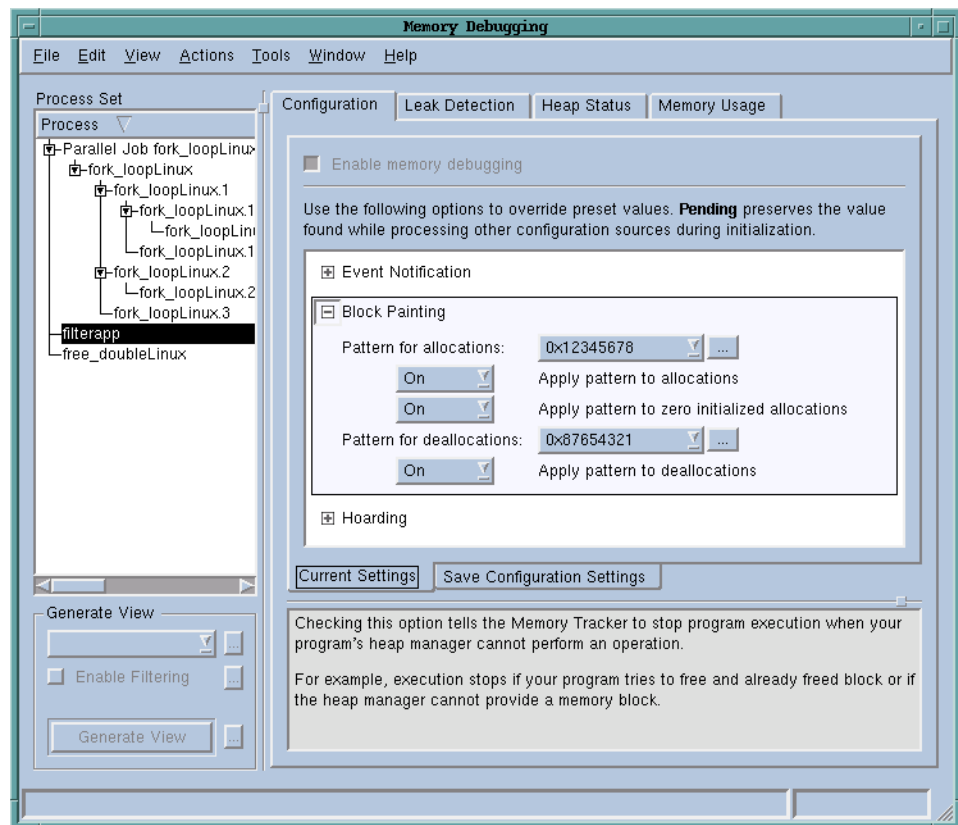
- **Block painting**, which tells the Memory Debugger to write a bit pattern into allocated and deallocated memory blocks.
- **Hoarding**, which tells the Memory Debugger to hold onto a memory block when the heap manager receives a request to free it. This is most often used to get beyond where a problem occurs. By allowing the program to continue executing with correct data, you sometimes have a better chance to find the problem. For example, if you also paint the block, it becomes easy to tell what the problem is. In addition, your program might crash. (Crashing while you are in TotalView is a good thing,

because TotalView will show the crash point. You immediately know where the problem is.)

- **Watchpoints**, which tell TotalView to stop execution when a new value is written into a memory block. If the Memory Debugger is painting deallocated blocks, you immediately know where your program freed the block.
- **Block tagging** (described in “Block Properties and Event Notification” on page 21), which tells TotalView to stop execution when your program deallocates or reallocates memory.

You enable painting and hoarding in the Memory Debugger Configuration Page.

Figure 24: Configuration Page



You can turn painting and hoarding on and off. In addition, you can tell the Memory Debugger what bit patterns to use when it paints memory. For more information, see “Block Painting” on page 31.

## Dangling Pointers

If you enable memory debugging, TotalView displays information in the Variable Window about the variable’s memory use. The following small program allocates a memory block, sets a pointer to the middle of the block, and then deallocates the block:

```
main(int argc, char **argv)
{
    int *addr = 0;      /* Pointer to start of block. */
    ...
}
```

```
int *misaddr = 0; /* Pointer to interior of block. */

addr = (int *) malloc (10 * sizeof(int));

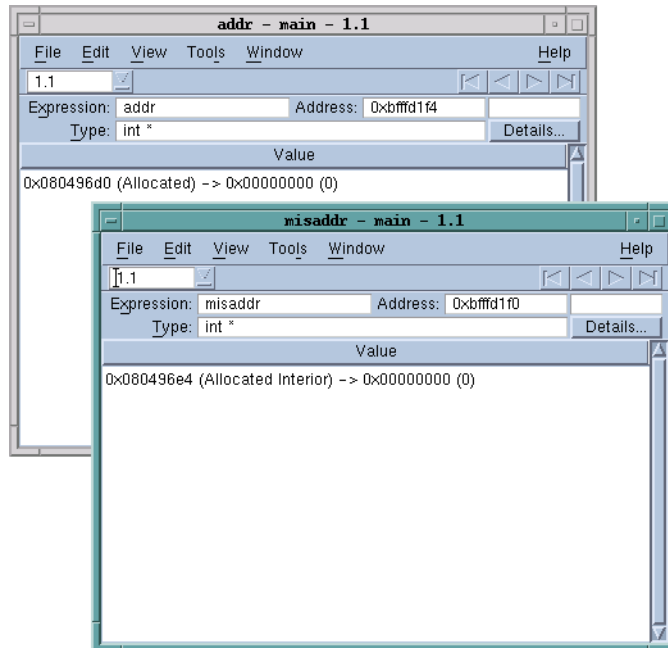
misaddr = addr + 5; /* Point to block interior */

/* Deallocate the block. addr and */
/* misaddr are now dangling. */

free (addr);
}
```

The following figure shows two Variable Windows. Execution was stopped before the **free()** function executed. Both windows contain a memory indicator saying that blocks are allocated.

Figure 25: Allocated Description in a Variable Window



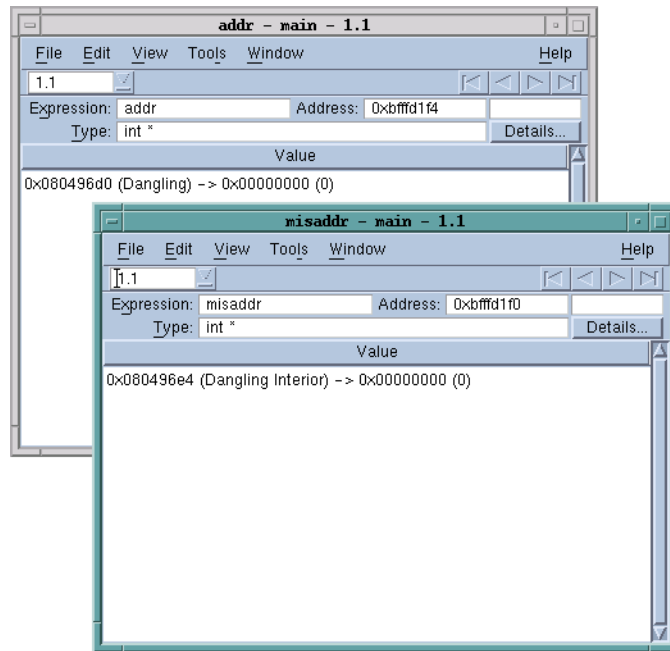
After your program executes the **free()** function, the messages change, as Figure 26 on page 29 shows.

## Examining Memory

So far, you've been reading about memory errors. If only things were this simple. The large amount of memory available on a modern computer and the ways in which an operating system converts actual memory into virtual memory may hide many problems. At some point, your program can hit a wall, thrashing the heap to find memory it can use or crashing because, while memory is available, the operating system can't find a block big enough to contain your data. In these circumstances, and many others, you can examine the heap to determine how your program is managing memory.



Figure 26: Dangling Description in a Variable Window



The Memory Debugger can display a lot of information, at times too much information. In all cases, you'll start by looking at what your program has done with the heap. You'll then be able to filter out information so to focus on issues.

Begin by displaying the Graphical View within the Heap Status Page. Here's how:

- 1 Select the Heap Status Tab.
- 2 Select the **Graphical View** item on the pulldown list in the Generate View area.
- 3 If you want the Memory Debugger to identify leaked memory in the display (and there's no reason that it shouldn't), select the ellipses (...) following this pulldown. This tells the Memory Debugger to display a preferences dialog box. In this box, check the **Label leaked memory blocks** item.
- 4 Press the **Generate View** Button.

The Memory Debugger responds by displaying a graphical view of the heap. (See Figure 27 on page 30.) If your program's heap is large, you may see a window telling you what kind of processing the Memory Debugger is performing. (See Figure 28 on page 30)

The display area has two parts. The upper contains many bars, each of which represents one allocation. The bar's color indicates if the memory block is allocated, deallocated, leaked, or within the hoard.

The bottom area has three divisions. The first contains a key to the colors used in the top area. In addition, it indicates how much memory is in each state. For example, the program used for this example has allocated 1620.78 KB of memory.

Figure 27: Heap Status Page: Graphical View

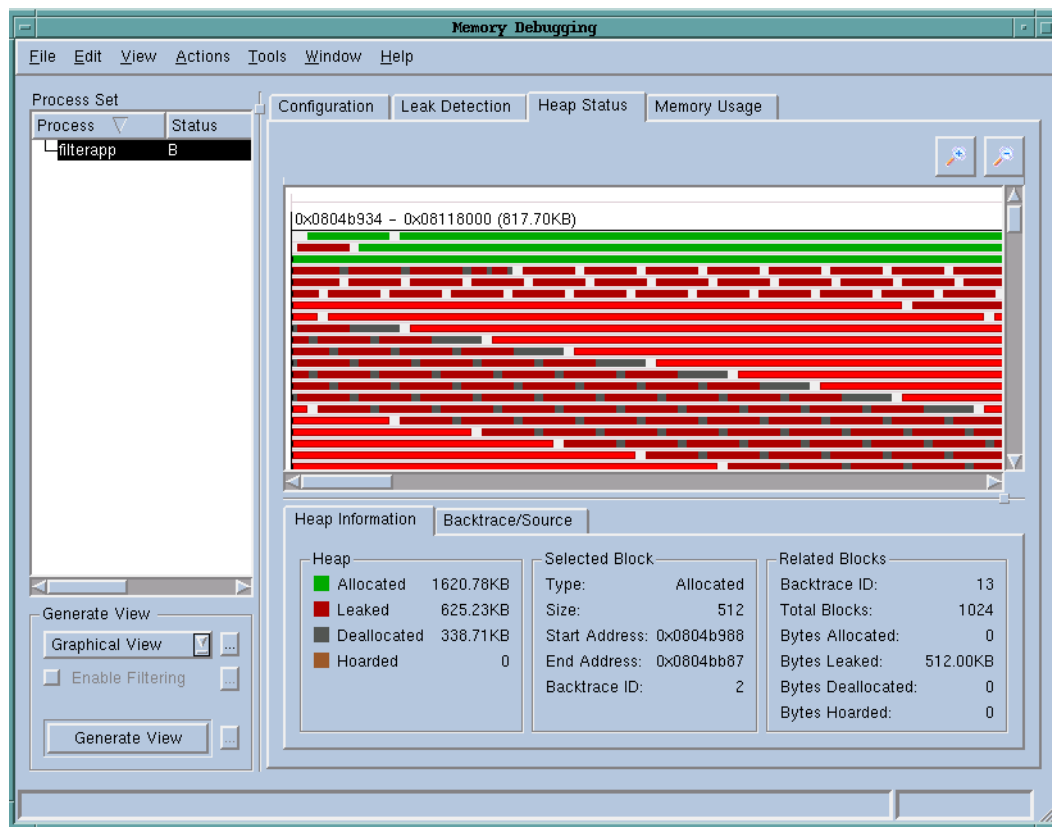
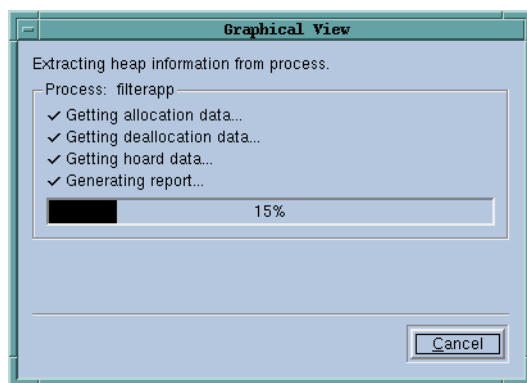


Figure 28: Creating the Graphical View Window



If you select a block, the center area contains information about this block. When you select a block, the Memory Debugger highlights it within the top area.

The right area lets you know how many other blocks were allocated from the same location. (Actually, this just shows how many allocations had the same backtrace. If your program got to the same place in different ways, they'd have different backtraces, so they wouldn't be considered related.)

Now that you have this information, you can begin making decisions. Obviously, you'd fix the leaks. If there were a lot of small blocks, is your program allocating memory too frequently? Should it be allocating memory in larger blocks and managing the allocated memory directly? Is there a pattern of allocations and deallocations that prevents reuse.

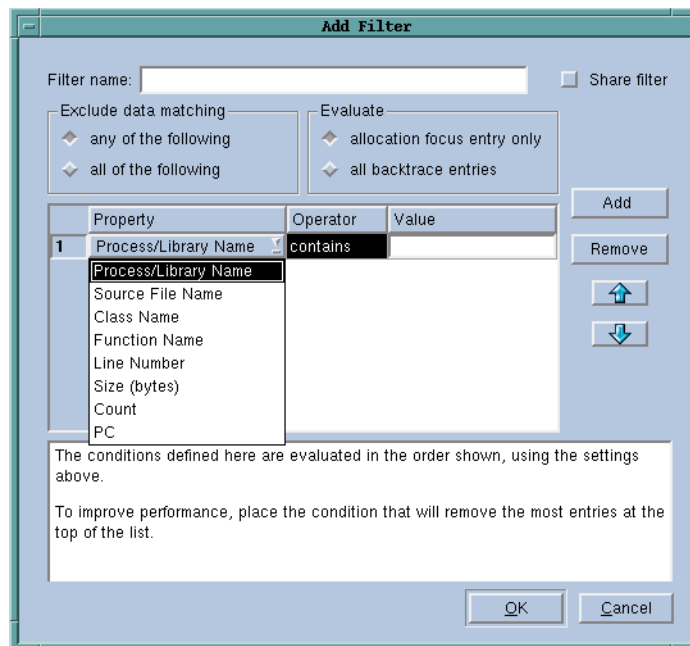


*Memory managers tend to be lazy. Unless they can easily reuse memory, they just get more. If you use the Memory Usage Page to monitor how your program is using memory, you'll probably find that your program only gets bigger. Once your program grabs memory from the operating system, it doesn't like to give it back. And, while it could reuse this memory if your program deallocates it, it is far easier and quicker to grab new memory.*

## Filtering

You can remove information from Backtrace and Source Views by adding a filter. For example, suppose you don't want the Memory Debugger to show blocks that are related to the `strdup()` function. By creating and applying a filter (see "Filtering" on page 38), the Memory Debugger will remove this information from the display. Here's an example of the dialog box you use to create a filter:

Figure 29: A Filter Dialog Box



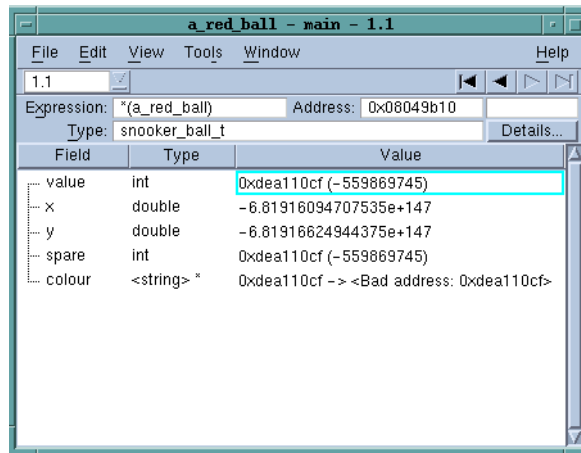
## Block Painting

When you enable block painting, TotalView paints a memory block with a bit pattern. You can either specify a pattern or use the default, as follows:

- The default allocation pattern is **0xa110ca7f**, which was chosen because it resembles the word "allocate".
- The default deallocation pattern is **0xdea110cf**, which was chosen because it resembles the word "deallocate". In most cases, you want TotalView to paint memory blocks when they are deallocated.

The following figure shows a variable whose memory was painted:

Figure 30: Block Painting



If the Memory Debugger paints memory for a variable that uses more memory than a word—for example, a double-precision variable—the value that TotalView displays in the Variable Window won't look like the paint pattern. For example, the value in an allocated memory block for a double-precision number is: `-6.81916624944375e-147`. You can, of course, cast the value to single precision if you are unsure if the value being displayed is your painting value.

Setting the allocation pattern lets you know if your program initialized a variable. For example, if you display the variable in a Variable Window and see the paint pattern, you'll immediately know that you have a problem.

If you also set a watchpoint on the memory block before your program deallocates it—you might only be able to set it on the first few words of the block—TotalView stops program execution just after the Memory Tracker paints it.

If you are setting a watchpoint on just one element of a structure or an array, you need to dive on the element so that it is the only item in the Variable Window. For example, if you want to set a watchpoint on the **colour** variable in the previous figure, dive on **colour**, and then select the **Tools > Watchpoint** command to set the watchpoint.

If you change the deallocation pattern while your program executes, the pattern lets you know when the block was deallocated. That is, because the Memory Debugger is using a different pattern after you change it, you will know if the memory was allocated or deallocated before or after you made the change.

If you are painting deallocated memory, you could be transforming a working program into one that no longer works. This is good as TotalView will be telling you about a problem.

## Hoarding

You can stop your program's memory manager from immediately reusing memory blocks by telling the Memory Debugger to hoard (that is, retain) blocks. Because memory blocks aren't being immediately reused, the data

within the blocks isn't being overwritten. This means that your program can continue running with the correct information even though it is accessing deallocated memory. If this weren't the case, any pointers into this memory block would be dangling. In some cases, this uncovers other errors, and these errors can help you track down the problem.

If you are painting and hoarding deallocated memory (and you should be), you might be able to force an error when your program accesses the painted memory.

The Memory Debugger holds onto hoarded blocks for a while before returning them to the heap manager so that the heap manager can reuse them. As the Memory Debugger adds blocks to the hoard, it places them in a first-in, first-out list. When the hoard is full, the Memory Debugger releases the oldest blocks back to your program's memory manager.

### Example 1: Finding a Multithreading Problem

When a multithreaded program share memory, problems can occur if a memory block is deallocated by one thread while it still being used by another. Because threads execute intermittently, problems are also intermittent. If you hoard memory, the memory will stay viable for longer because it cannot be reused immediately.

If intermittent program failures stop occurring, you know what kind of problem exists.

One advantage of this technique is that you can relink your program (as is described in Chapter 4, "*Creating Programs for Memory Debugging*," on page 83) and then run TotalView and the Memory Debugger against a production program that has not been compiled using `-g` compiler debugging option.

If you don't know where the problem occurs, you will probably need to increase the number of blocks being hoarded and the hoard size.

### Example 2: Finding Dangling Pointer References

Hoarding is most often used to find dangling pointer references. Once you know the problem is related to a dangling pointer, you need to locate where the memory is deallocated. One technique is to use block tagging (see "*Block Properties and Event Notification*" on page 21). Another is to use block painting to write a pattern into deallocated memory. If you also hoard painted memory, the heap manager will not be able to reallocate the memory.

If the memory was not hoarded, the heap manager could reallocate the memory block. When it is reallocated, a program can legitimately use the block, changing the data in the painted memory. If this occurs, the block is both legitimately allocated and its contents are legitimate in some context. However, the older context has been destroyed. Hoarding delays the recycling of the block. In this way, it extends the time available for you to detect that your program is accessing deallocated memory.



# Using the Memory Debugger Window

## 2

This chapter examines the Memory Debugger Window. It includes the following topics:

- “*About the Memory Debugger*” on page 35
- “*Common Operations*” on page 37
- “*Configuration Page*” on page 44
- “*Leak Detection Page*” on page 52
- “*Heap Status Page*” on page 56
- “*Memory Usage Page*” on page 60

## About the Memory Debugger

---

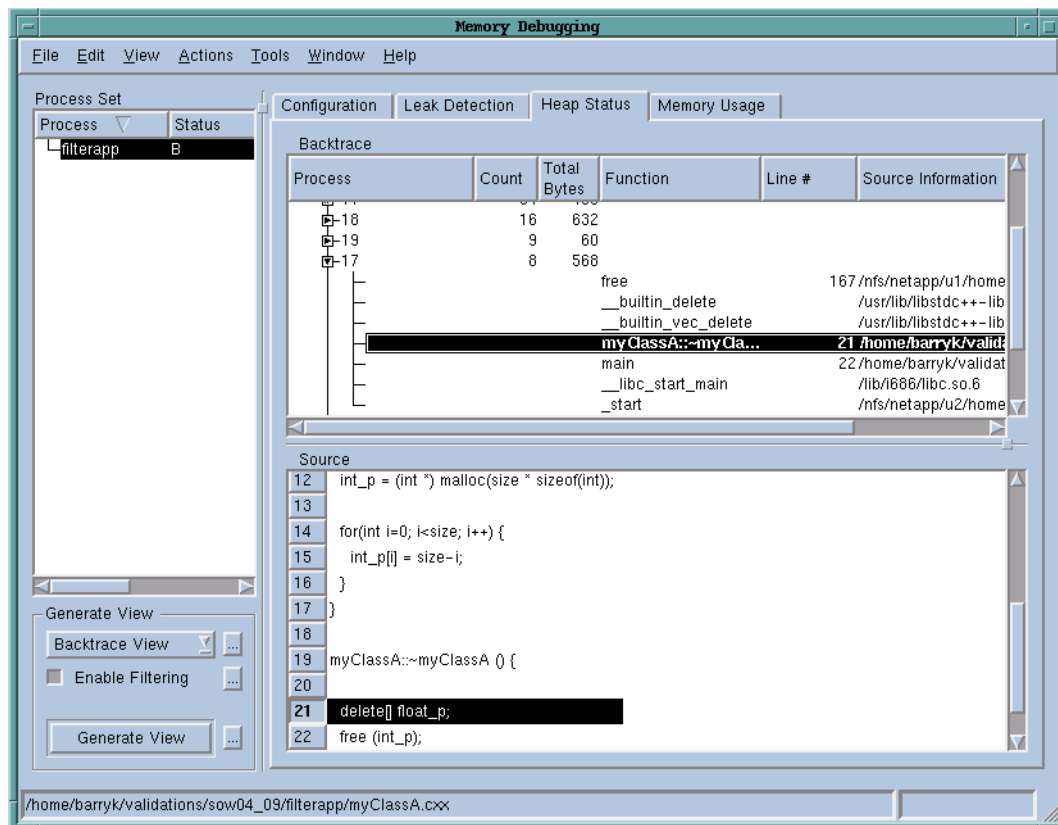
When you configure the Memory Debugger or display a view, the action that the Memory Debugger takes is based on the processes that you select on the left side of the window. (The figure on the next page shows this window.)

The controls in the **Generate View** area tell the Memory Debugger which view to create on the right side of the window. This information is called a *view* because the Memory Debugger just shows a part of the information contained in the Memory Debugger tracking agent. (For information on this agent, see “*Behind the Scenes*” on page 5.)

### Process Set Selection

Configuring the Memory Debugger tells it which processes to track and what actions to perform. For example, the Memory Debugger Window shown on the next page can track more than one program. One of these programs has more than one process. If you select three processes out of the nine processes in this window, a leak detection view only shows leaks from these three processes. It ignores leaks in other processes.

Figure 31: The Memory Debugger



Be careful how many processes you select. With large multiprocess programs, you might be asking the Memory Debugger to process and analyze an enormous amount of data. In most cases, if you select one or two significant processes, you'll receive the information you need. Although the process of generating a view is lengthy, you can redisplay the information quickly after the Memory Debugger creates it.

### Generate View Area

When you are viewing any page except the Configuration Page, you must tell the Memory Debugger which view it should display. (Specifying a view tells the Memory Debugger how it should display its information.) The controls in this area of the window are as follows:

#### Pulldown list

Select a view from this list. Clicking on the arrow on the right side of this list displays your choices. This pulldown is not active when the Memory Debugger is displaying the Configuration Page.



Click this button to display a dialog box that contains preferences that modify or affect a view. The discussions of those page in other sections of this chapter describes these preferences.



**Enable Filtering** Selecting this check box tells the Memory Debugger to apply filters to the information it is displaying. For additional information, see “*Filtering*” on page 38.



Click this button to display the **Data Filters** Dialog Box. For more information, see “*Filtering*” on page 38.

**Generate View** After you select a view, pressing this button tells the Memory Debugger to display it.



If you need to save the information contained within a view, select this button. The Memory Debugger responds by displaying a dialog box that lets you write this information to disk. For more information, see “*Saving Views*” on page 41.

## Common Operations

### Rows and Columns

If a page displays information in columns, you can resize columns, change the column order, and control which columns the Memory Debugger displays, as follows:

- To resize a column, place the mouse pointer over the vertical column separator in the header. Press your left mouse button and drag the separator so that you’ve made the column as wide or as narrow as you want it to be. After you finished dragging the separator, release the left mouse button. The following figure shows the second column being made wider (and the first being made smaller):

Figure 32: Resizing

Bytes	Count	End Address	Begin Address	Backtrace ID	Flags
-------	-------	-------------	---------------	--------------	-------

If you double-click on a separator, the Memory Debugger readjusts all widths.

- To change the column order, place your mouse pointer in a column header, press your left mouse button, and then drag the column to its new position. After it is in its new position, release the left mouse button. In the following example, the **Begin Address** column is being moved to the left:

Figure 33: Changing Position

Process	Bytes	Count	Begin Address	End Address	Backtrace ID	Flags
---------	-------	-------	---------------	-------------	--------------	-------

- To tell the Memory Debugger to hide a column or display a column you previously hid, right-click anywhere in the column header area. From the displayed context menu, click on an entry. If the entry is hidden, the

Memory Debugger displays it. If the column is displayed, the Memory Debugger hides it. The following figure shows this context menu:


Figure 34: Displaying and Hiding Columns

Process	Bytes	Count	End Address	Begin Address	Backtrace	Flags
global_leak	450	9				
global...	450	9				
main	450	9				
L	450	9				
	90	1	...4519122	...45191		one
	80	1	...4519024	...4518:		one
	70	1	...4518934	...4518:		one
	60	1	...4518860	...4518:		one

- To tell the Memory Debugger to sort a column, click on the column heading. You can only sort some columns.


## Filtering

The amount of information that the Memory Debugger displays when you ask for a Leak Detection or Heap Status View can be considerable. In addition, this information includes memory blocks allocated within any shared library that your program uses. In other cases, your program may be allocating memory in many different ways and you only want to focus on a few of them. You can eliminate information from a backtrace or source view by using a filter. Filtering is a two-step process:

- 1 Create a filter by selecting the  button that is to the right of the **Enable Filtering** check box within the **Generate View** area. You can also use the **Tools > Filter** command.
- 2 At a later time, select the **Enable Filtering** check box.

When filtering is enabled, the Memory Debugger looks at each enabled filter and applies it to the view's data. In addition, each can have any number of actions associated with it.

### Adding, Deleting, Enabling and Disabling Filters

After you select the  button that is to the right of the **Enable Filtering** check box, the Memory Debugger displays a dialog box that allows you add, delete, enable, delete, and change the order in which the Memory Debugger applies filters. (See Figure 35 on page 39)

The controls within this dialog box are as follows:

#### ☒ Enable and Disable

When checked, the filter is enabled.

#### Add

After pressing this button, the Memory Debugger displays the **Add Filter** Dialog Box. Using that dialog box, you can define one filter. That dialog box will be discussed later in this section.

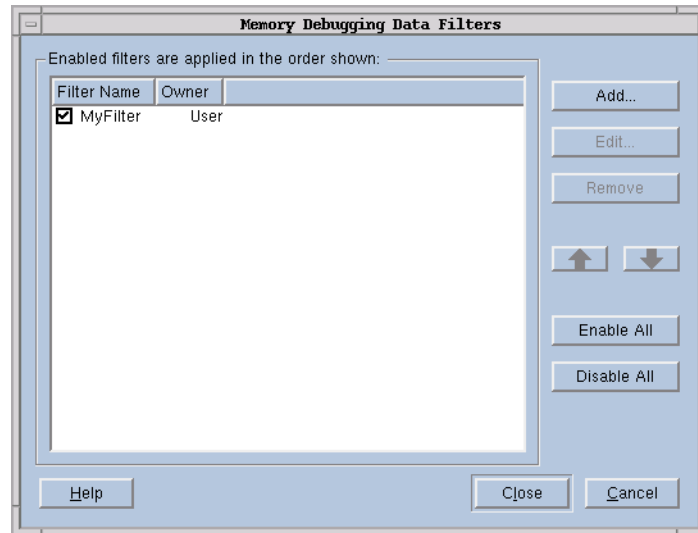
#### Edit

Displays a dialog box that allows you to change the selected filter's definition. The displayed **Edit Filter** Dialog Box is identical to the **Add Filter** Dialog Box.

#### Remove

Deletes the selected filter.

Figure 35: Memory Debugging Data Filters Dialog Box



#### Up and Down

Moves a filter up or down in the filter list. As the Memory Debugger applies filters in the order in which they appear in this list, you should place filters that remove the most entries at the top of the list. As filtering can be a time-consuming operation, this can increase performance.

**Enable All** Enables (checks) all filters in the list.

**Disable All** Disables (unchecks) all filters in the list.

### Adding and Editing Filters

After you select the **Add** button within the **Memory Debugging Data Filters** Dialog Box, the Memory Debugger displays the **Add Filter** Dialog Box. (See Figure 36 on page 40.)

Selecting the **Edit** button within the **Memory Debugging Data Filters** Dialog Box tells the Memory Debugger to display a nearly identical window.

The controls within this window are as follows:

**Filter name** Enter the name of the filter. This name will appear in the **Memory Debugging Data Filters** Dialog Box.

**Share filter** Selecting this button tells the Memory Debugger that the filter you are creating will be shared. *Shared* means that anyone using TotalView can apply the filter.

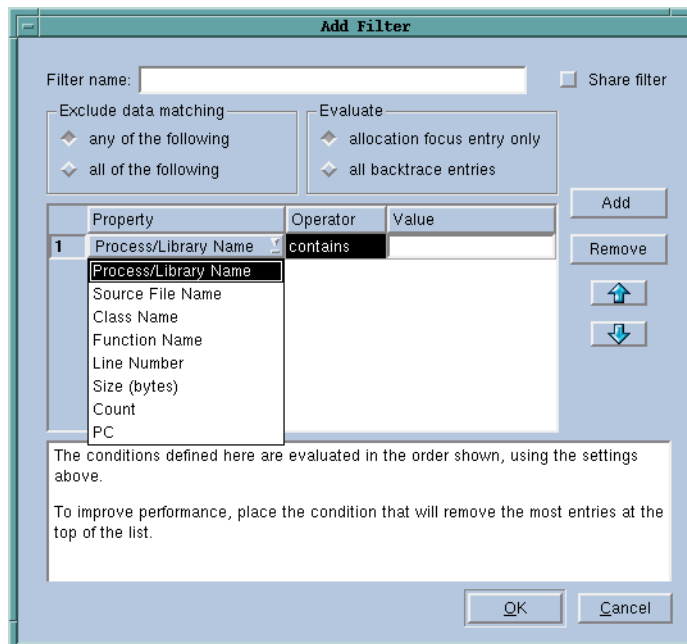


*This button only appears if you have write permissions for the TotalView lib directory.*

**Add** Pressing this button tells TotalView to add a blank line beneath the last criterion in the list. You can now enter information defining criterion within this new line.

**Remove** Deletes the selected criterion. To select a criterion, select the number to the left of the definition.

Figure 36: Add Filter Dialog Box:  
Showing Properties



**Up and Down** Changes the order in which criteria appear in the list. While changing the order doesn't change the results of the filtering operation, placing criteria that exclude the most information at the top of the list improves performance.

### Exclude data matching

If you have more than one criterion, the selected radio button indicates if *any* or *all* of the criteria have to be met.

#### any of the following

When selected, a memory entry is removed when the entry matches any of the criteria in the list.

#### all of the following

When selected, a memory entry is only removed if it fulfills all of the criteria.

### Evaluate

When evaluating a filter, you can limit which backtraces the Memory Debugger looks at.

#### allocation focus entry only

When selected, tells the Memory Debugger that it should remove the entry only if the criteria you set is valid on an entry that is also the allocation focus.

The allocation focus is the point in the backtrace where the Memory Debugger believes your code called **malloc()**.

For example, if you define a filter condition that says **Function Name contains my\_malloc** and set this entry to **allocation focus entry only**, the Memory Debugger only removes blocks whose allocation focus contains

**my\_malloc.** That is, it only removes blocks that were allocated directly from **my\_malloc**.

In contrast, if you set this entry to **all backtrace entries**, the Memory Debugger removes all blocks that contain **my\_malloc** anywhere in their backtrace.

#### all backtrace entries

When selected, the Memory Debugger applies filter criteria to all function names within the backtrace.

#### Criteria

A filter is made up of criteria. Each criterion has three parts: a property, an operator, and a value. That is, you can indicate what the Memory Debugger looks for. For example, you can look for a Process/Library Name (the *property*) that contains (the *operator*) **strdup** (the *value*).

#### Property

When evaluating an entry, the Memory Debugger can look at one of eight properties for one criterion. (See Figure 36 on page 40.) Select one of the items from the pulldown list. These items are:

Process/Library Name  
Source File Name  
Class Name  
Function Name  
Line Number  
Size (bytes)  
Count  
PC

#### Operator

The operator indicates the relationship the *value* has to the *property*. (See Figure 37 on page 42.) Select one of the items from the pulldown list. If the property you've selected is a string, the Memory Debugger displays the following list:

contains  
not contains  
starts with  
ends with  
equals  
not equals

If the item is numeric, it displays the following list:

<=  
<  
=  
!=  
>  
>=

#### Value

Type a string or a number that indicates what is being compared.

## Saving Views


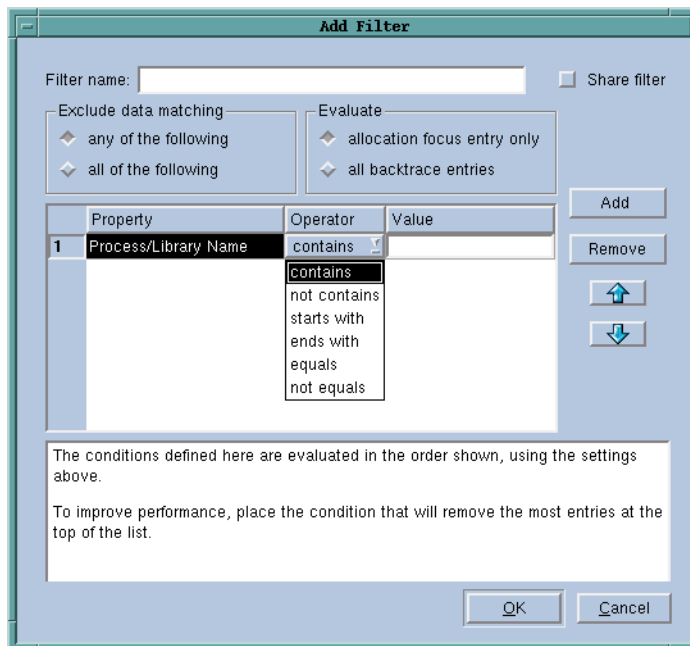
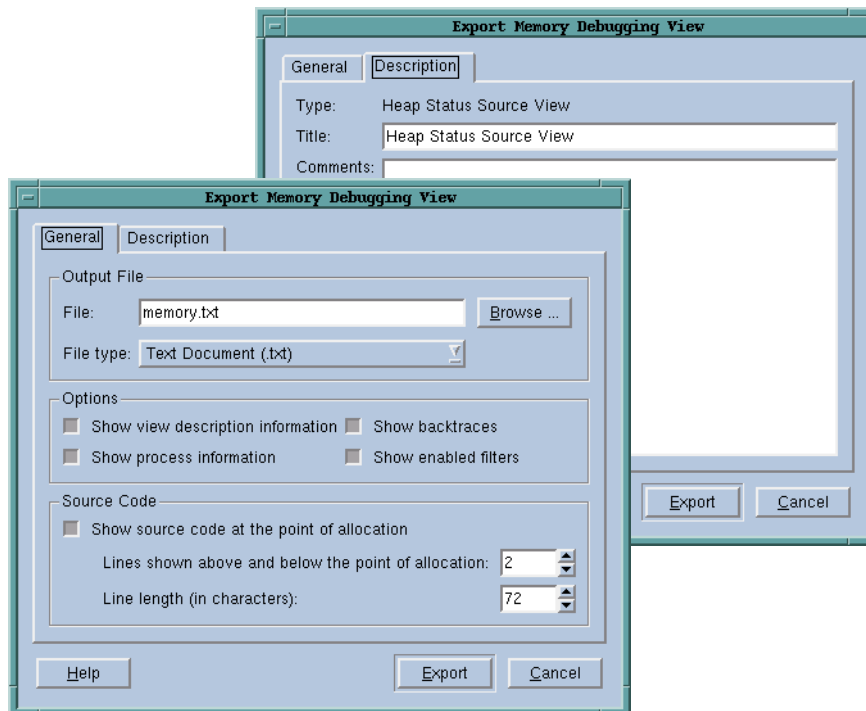
If you need to write view information to disk, press the  button, which is immediately to the left of the **Generate View** button. The Memory Debugger

Figure 37: Add Filter Dialog Box: Showing Operators



responds by displaying a dialog box with two tabs. Both tabs are shown in the following illustration:

Figure 38: Saving Views



### General Page

The General Page contains the controls that let you specify what you want written. Here is what these controls do:

<b>Output File</b>	The controls within this area tell the Memory Debugger where it should write memory information.
<b>File</b>	Enter the name of the file being created. You can change this from its default value by editing the text.
<b>Browse</b>	Press this button to display a dialog box that lets you select the directory in which the Memory Debugger will write the file.
<b>File type</b>	Select a file type, At version 6.7, the only format you can select is text.
<b>Options</b>	The controls within this area tell the Memory Debugger what additional information it should write into the file.
	<b>Show view description information</b> When selected, the Memory Debugger writes information about the view type, data displayed, the user creating the file, and the host, date, and the comment recorded in the Description Page.
	<b>Show process information</b> When selected, the Memory Debugger writes information about the processes that were selected when you generate the view.
	<b>Show backtraces</b> When selected, the Memory Debugger writes stack backtrace information for the memory allocations in the view. If the view being displayed already contains backtraces, the Memory Debugger ignores this option.  Selecting this option increases the time the Memory Debugger needs to create the report. In addition, the size of the created file will be much larger.
	<b>Show enabled filters</b> When selected, the Memory Debugger names and describes the filters it used when it generated the view.
<b>Source Code</b>	The Memory Debugger can also display lines from your source code.
	<b>Show source code at the point of allocation</b> When selected, the Memory Debugger displays source code information.
	<b>Lines shown above and below the point of allocation</b> Tells the Memory Debugger how many lines of source code above and below the allocation statement should also be displayed.
	<b>Line length (in characters)</b> Tells the Memory Debugger how many characters it should use in each line when displaying information. Lines that are longer than this length are truncated.

## Configuration Page

### Description Page

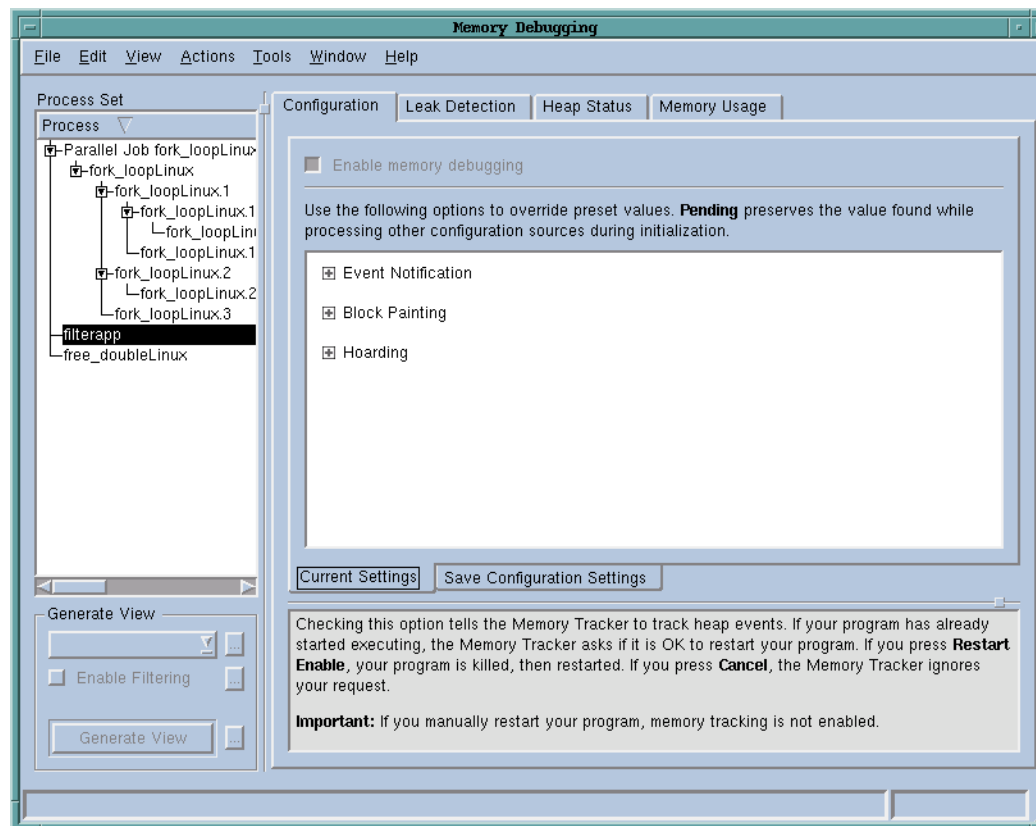
If you are writing a number of files, adding comments can help you identify the report. You can enter the following information:

<b>Title</b>	If the default title isn't what you want, enter something more descriptive here.
<b>Comments</b>	Enter text that describes the view information being written to disk.

## Configuration Page

The controls on the Configuration Page direct the actions that the Memory Debugger performs. They also allow you to save and restore settings that you have saved to disk. The following figure shows this page:

Figure 39:  
Configuration  
Page



### Current Settings Page

The current settings page is where you tell the Memory Debugger which actions it should take when memory events occur. In addition, you can tailor these actions to your needs.



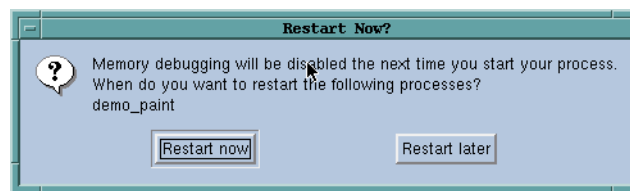
*While you must explicitly tell the Memory Debugger to track your program's use of the heap API, you do not need to enable memory debugging to obtain a Memory Usage View.*



The **Enable memory debugging** check box tells the Memory Debugger if it should track your program's use of the heap API. If TotalView can dynamically enable memory debugging, selecting this button loads the Memory Debugger. Most computing architectures do allow TotalView to enable the Memory Debugger before your program begins executing. However, TotalView cannot directly enable programs that run on an IBM RS/6000 or which run remotely. See Chapter 4, "Creating Programs for Memory Debugging," on page 83 for more information.

You cannot enable or disable the Memory Debugger while your program is executing. If you try, the Memory Debugger opens a dialog box asking if it should restart your program.

Figure 40: Restart Now Dialog Box

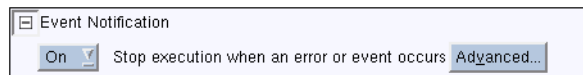


The third line of this error message has the name of the program or process that must be restarted.

### Event Notification

If a memory event occurs using a function within the heap API, the Memory Debugger can tell TotalView to stop the program's execution so that you can determine the source of the event. For more information, see "Finding *free()* and *realloc()* Problems" on page 17.

Figure 41: Memory Error Notification Area



Here is a description of the controls in this section:

#### Stop execution when an event or error occurs

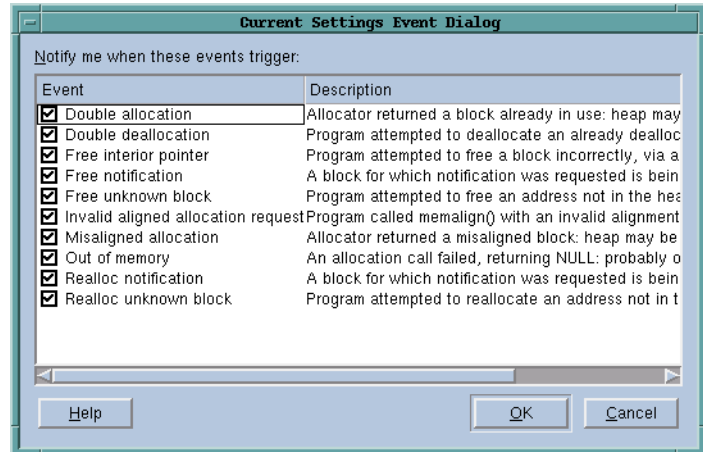
Checking this box tells the Memory Debugger to stop program execution and display a dialog box when it detects that an event occurred that is related to using the heap API.

You can turn notification on and off both before and while your program is executing.

#### Advanced

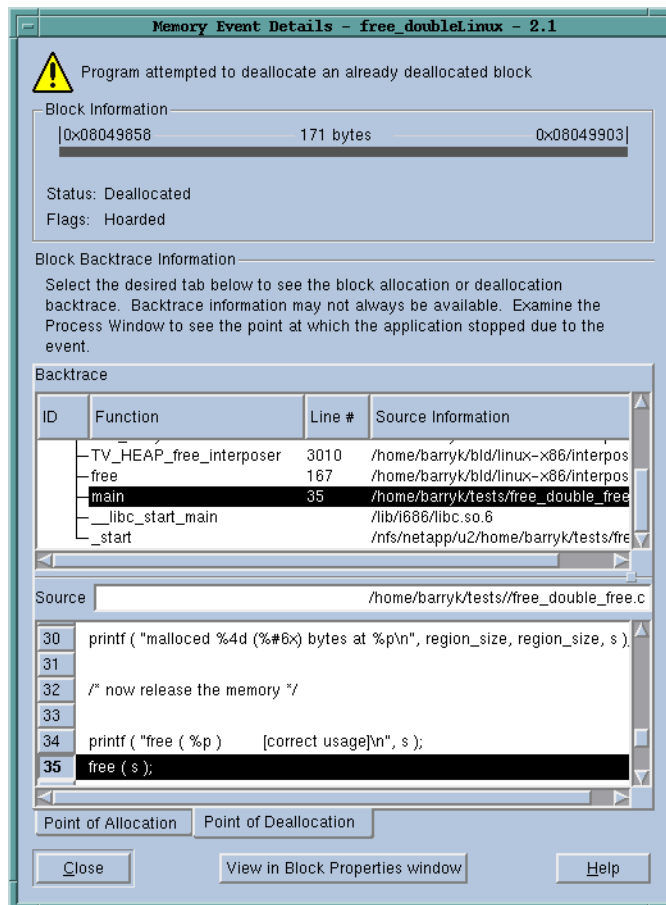
Selecting this button tells the Memory Debugger to display a dialog box from which the events for which the Memory Debugger will stop execution. (See Figure 42 on page 46. By default, notification occurs for all events. You can individually turn an event off if you need to.)

Figure 42: Current Settings Event Dialog Box



When an event occurs, the Memory Debugger stops program execution and tells TotalView to display its **Memory Event Details** Window. (See Figure 43 on page 46.)

Figure 43: Memory Error Details Window



This window has four areas, as follows:

- The top line tells you what type of error or event occurred.
- The **Block Information** area gives the memory location of the block and its status.
- The third area contains the function backtrace if the error or event is related to a block allocated on the heap. The Memory Debugger retains information about the backtrace that existed when the memory block was allocated and the backtrace when it was deallocated. You can tell the Memory Debugger which it should display by selecting either the **Point of Allocation** or **Point of Deallocation** tab.  
If a memory error occurred, the deallocation backtrace is often the same as the backtrace being shown in the Process Window's Source Pane. If the memory error occurs after your program deallocated this memory, the backtraces are different.
- The bottom area shows you where the allocation or deallocation occurred in your program.



*In some cases, the Memory Debugger does not display an allocation backtrace. For example, if you try to free memory allocated on the stack or in a data section, there's no backtrace because your program did not allocate the memory.*

If you need to redisplay the Memory Block Window after you dismiss it, select the **Tools > Memory Event Details** command.

### Memory Block Properties Window

You can obtain additional information about the block associated with an event if you press the **View in Block Properties window** button that is at the bottom of the **Memory Event Details** Window. (See Figure 44 on page 48.)

The information in this window is a combination of what can be displayed in other views. For example, the bottom portion is similar to a Source View displayed in a Heap Status view. Some of the top portion is what you will see in a Heap Status Graphical View. If the block is associated with an event, this information is also displayed.

You'll find more information about this window in "*Block Properties and Event Notification*" on page 21 and in the online help.

### Block Painting

When you enable memory block painting, the Memory Debugger writes a bit pattern into newly allocated and newly deallocated heap memory blocks. For information on using block painting, see "*Block Painting*" on page 31. (Figure 45 on page 48 shows block painting controls.)

Here is a description of these controls:

#### Pattern for allocations

The Memory Debugger uses the bit pattern in this box when it paints heap memory that was just deallocated. It uses the same pattern for normal allocations and zero-initialized allocations, which are allocations created by functions such as **calloc()**. The pulldown list contains patterns that you used previously.

Figure 44: Memory Block Properties Window

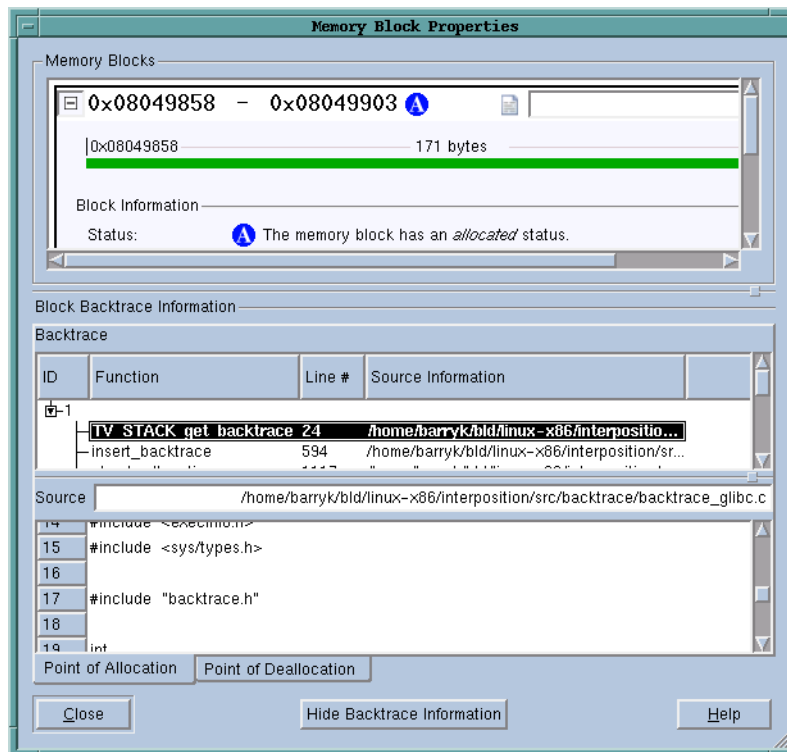
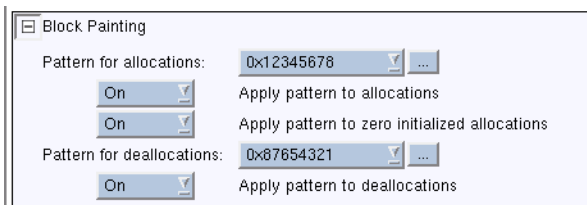


Figure 45: Memory Block Painting Area




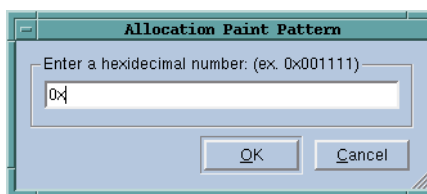
When you click the  button to the right of the pattern pulldown list, the Memory Debugger displays a dialog box into which you can type a new pattern:

Figure 46: Allocation Paint Pattern Dialog Box



If your program has not started executing, the Memory Debugger might not be able to display a pattern. If it cannot display a pattern, it displays **<pending>**. You can change this pattern at any time and as many times as you want while your program is executing.

Changing the pattern can help you identify when your program allocated a memory block. For example, when you see a pattern, you can tell if it was painted before or after you made a change.

If a data value uses more bits than indicated by the paint pattern, TotalView interprets the value using the number of bytes that the variable uses, not the number of bytes in the paint pattern. This means that you might need to cast the displayed value.

If you uncheck this box, the Memory Debugger stops painting allocated memory. You can recheck this box at a later time without having to restart your program.

#### Apply pattern to allocations

When **On** is selected, the Memory Debugger paints allocated memory using the bit pattern shown in the **Pattern for allocations** text field.

#### Apply pattern to zero initialized allocations

When **On** is selected, the Memory Debugger paints allocated memory that is set to zero by calls such as **calloc()** using the bit pattern shown in the **Pattern for allocations** text field.

You cannot paint zero-allocated memory unless you are also painting normal allocations. If you set the **Apply pattern to allocations** to **Off**, the Memory Debugger also sets this control to **Off**.



*Setting this option to On can break your program if you depend upon the allocated memory being set to zero.*

#### Pattern for deallocations

The Memory Debugger uses the bit pattern in this box when it paints newly deallocated heap memory. For more information, see "Pattern for allocations" on page 47.

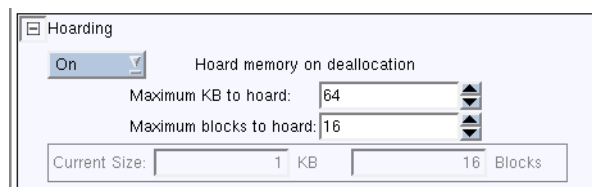
#### Apply pattern to deallocations

When **On** is selected, the Memory Debugger paints deallocated memory using the bit pattern shown in the **Pattern for deallocations** text field.

### Hoarding

The Memory Debugger can delay handing freed memory back to the heap manager. This is called *hoarding*. For more information, see "Hoarding" on page 32.

Figure 47: Memory Hoarding Area



Here is a description of these controls:

#### Hoard memory on deallocation

When **On** is selected, the Memory Debugger hoards memory. You can change this value while your program is executing.

If you set this value to **Off** while your program is executing, the Memory Debugger no longer hoards newly deallocated blocks. It does not, however, release blocks that it previously retained.

If the hoard is full and the Memory Debugger needs to hoard a new block, it releases the oldest blocks (that is, those that it first hoarded) so there's enough room in its hoard buffer. You can change the size of the hoard using the next two controls.

#### Maximum KB to hoard

By default, the hoard can grow to 256 KB. You can change the hoard's buffer size by changing this value.

#### Maximum blocks to hoard

By default, the hoard can contain up to 32 memory blocks. You can change the number of blocks by changing this value.

The gray area underneath these controls indicates the **Current Size** of the hoard. You are told how many kilobytes the hoard is using and how many different blocks are contained within it.

### Save Configuration Page


The Save Configuration Page (see Figure 48 on page 51) contains four sets of controls. The first three, **Event Notification**, **Block Painting**, and **Hoarding** are the same as the controls within the Current Settings Page and have already been discussed in this topic. The fourth, **Logging**, is new.



*Logging can only be specified in a saved file that is read when TotalView is initialized. You cannot specify logging interactively.*

The controls in this area are:

#### Log Memory Debug Information

When set to **On**, the Memory Debugger writes its information to stdout, stderr, or to a file. You can edit the file name. Select the  button to name the directory into which the Memory Debugger writes information. By default, it writes information into the program's directory.

#### Log all allocations on exit

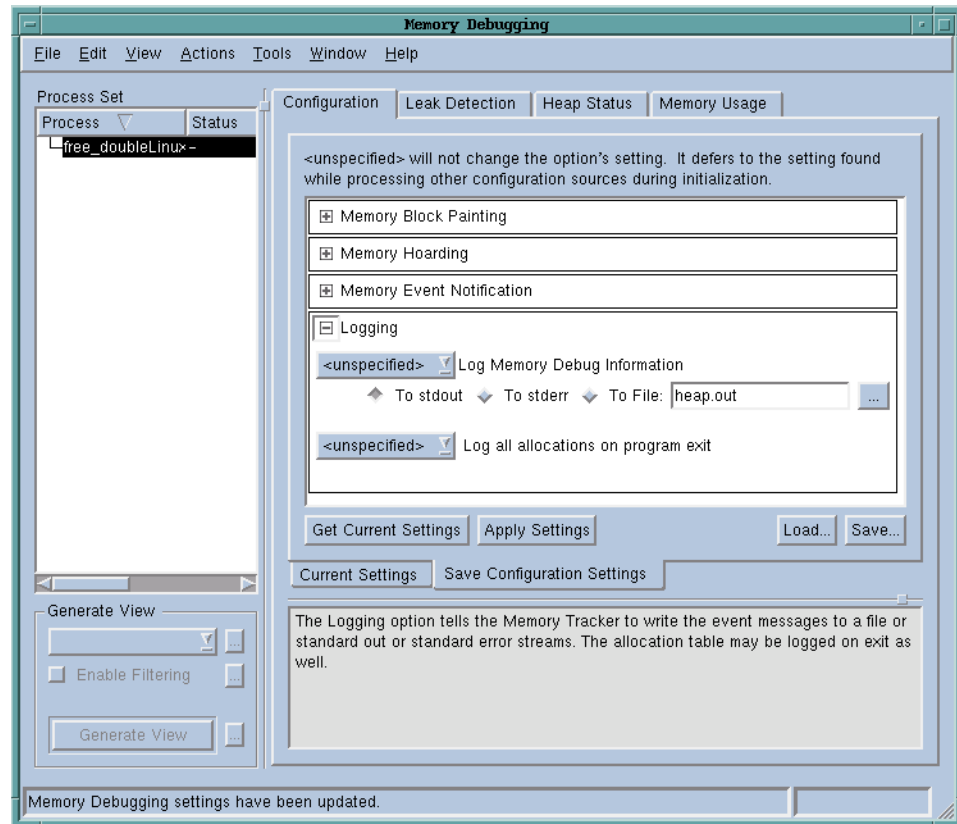
When set to **On**, the Memory Debugger writes allocation information to the location set in the **Log Memory Debug Information** command.

The four commands at the bottom are as follows:

#### Get Current Settings

Sets the controls within this page to be the same as those that set on the **Current Settings** Page.

Figure 48: Configuration Page

**Apply Settings**

Sets the controls on the **Current Settings Page** to be the same as those on this page.



*This command ignores changes that occur within the Logging area. Logging can only be enabled if it is enabled in a default.hiarc file contained in your current directory or in your .totalview/hia directory. If your configuration file has another name or is stored elsewhere, you must type the file's name and location in the TVHEAP\_ARGS variable.*

**Load**

Reads a saved configuration file and sets the controls on this page to those values. After loading configurations, you still need to use the **Apply Settings** command to make them active.

After pressing this button, the Memory Debugger displays an explorer window that you can use to locate the file you want to load.

**Save**

Writes the configuration displayed in this page to a file. After pressing this button, the Memory Debugger displays an explorer window that you can use to locate the directory into which you want to write the file. You can also use the explorer window to enter a name for this file.

### Presetting the Memory Debugger

The Memory Debugger gives you several ways in which you can preset values so that they do not have to be set in the Memory Debugger. The following list explains the places where you can preset values:

- 1 After writing a configuration file, you can specify that TotalView read the values in automatically. In order, it looks in three places: (1) the `TVHEAP_CONFIG_FILE` environment variable, (2) a file named `default.hiarc` contained within the current directory, and (3) a file named `hia/default.hiarc` contained within your `.totalview` subdirectory.
- 2 You can specify values using the `TVHEAP_ARGS` environment variable. For more information, see "Using the `TVHEAP_ARGS` Variable" on page 90.

## Leak Detection Page

---

The Memory Debugger can display information about the leaks it discovers in two ways: using a Source View or a Backtrace View. Each view displays approximately the same information.



*Be careful how many processes you select. With large multiprocess programs, you might be asking the Memory Debugger to process and analyze an enormous amount of data. In most cases, if you select one or two significant processes, you'll receive the information you need. Although the process of generating a view is lengthy, you can redisplay the information quickly after the Memory Debugger creates it.*

### Source View

The Source View organizes the leaks in your program by the program, routine, file, and block.

To create this view:

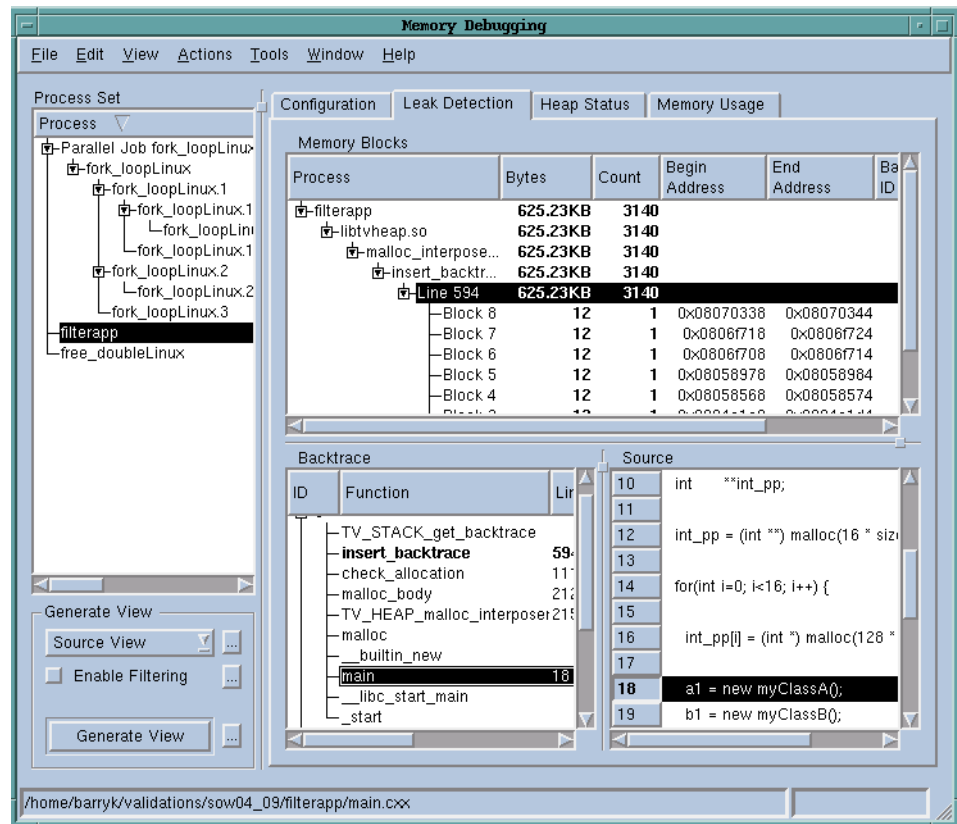
- Select the processes for which you want information in the **Process Set** area.
- Select **Source View**, and then select **Generate View**.

In this view, the first column, **Process**, contains a hierarchical display organizing your program's information. The Backtrace and Source Panes contain additional information about the line you select in the Memory Blocks Pane. In other words, this view organizes the information in the same way that your program is organized.

Figure 49 on page 53 shows a Source View. In this figure, the bottom-most rows in the hierarchy contain information about an individual leak. As you go up the tree towards the process name, the Memory Debugger summarizes the number of bytes and the number of leaks associated with the information at lower levels of the tree. In this example, the program leaked 625.23 KB and 3,140 allocations were associated with leaks.



Figure 49: Leak Detection  
Page: Source View

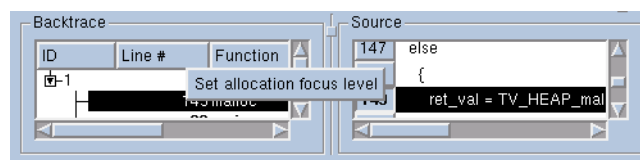


This explanation and the figure underemphasize the leak summary. Programs do leak memory. It is usually not practical to fix all leaks. If you click on the Bytes columns, the Memory Debugger sorts the table so that you can see what locations are leaking the most memory. This lets you focus on places leaking the most memory.

When you click on a line in the Memory Blocks Pane, the Memory Debugger shows information in the Backtrace Pane, as follows:

- The backtrace being displayed is the one that existed when your program allocated the memory block. The Memory Debugger highlights the frame that it thinks is the one you should be focusing on. That is, it highlights where the memory allocation was made. If it guesses wrong, you can reset the hierarchy of backtraces by right-clicking your mouse on the backtrace that you want displayed, as follows.

Figure 50: Backtrace and Source  
Panels



From the context menu, select **Set allocation focus level**.

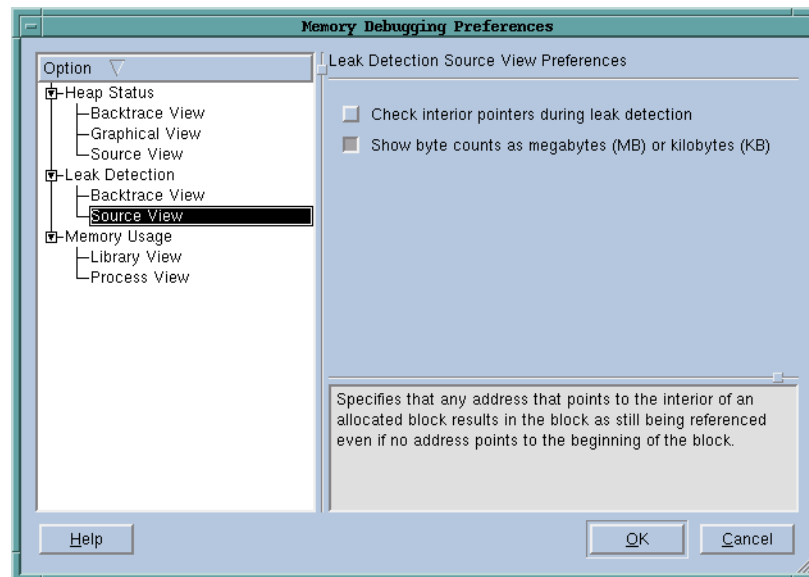
For example, assume that you have created a function named **my\_malloc()** that filters all of your memory allocations. The Memory


Debugger would probably guess that this is the function to highlight in the Backtrace Pane. However, you probably want to set the allocation focus on the function that called `my_malloc()`. Do this by selecting that function, and then right-clicking on it to invoke the **Set allocation focus level** command.

- The Source Pane shows the line in your program that contained the memory allocation statement. When you click on a backtrace ID, the Memory Debugger updates the Source Pane to show the line. The line number associated with this line is the same line number that appears in the Process Window Source Pane.

You can set two preferences for Leak Detection views. After displaying the preferences dialog box, the Memory Debugger displays the following dialog box:

Figure 51: Leak Detection Source View Preferences



To set preferences associated with the Source View, select the  button within the Generate View area on the left. The preferences are as follows:

### Check interior pointers during leak detection

Tells the Memory Debugger to consider a block as being referenced if a pointer is pointing anywhere within the block instead of just at the block's starting location. In most programs, the code should be keeping track of the block's boundary. However, if your C++ program is using multiple inheritance, you may be pointing into the middle of the block without knowing it.

Use this option with some caution as it can affect performance.

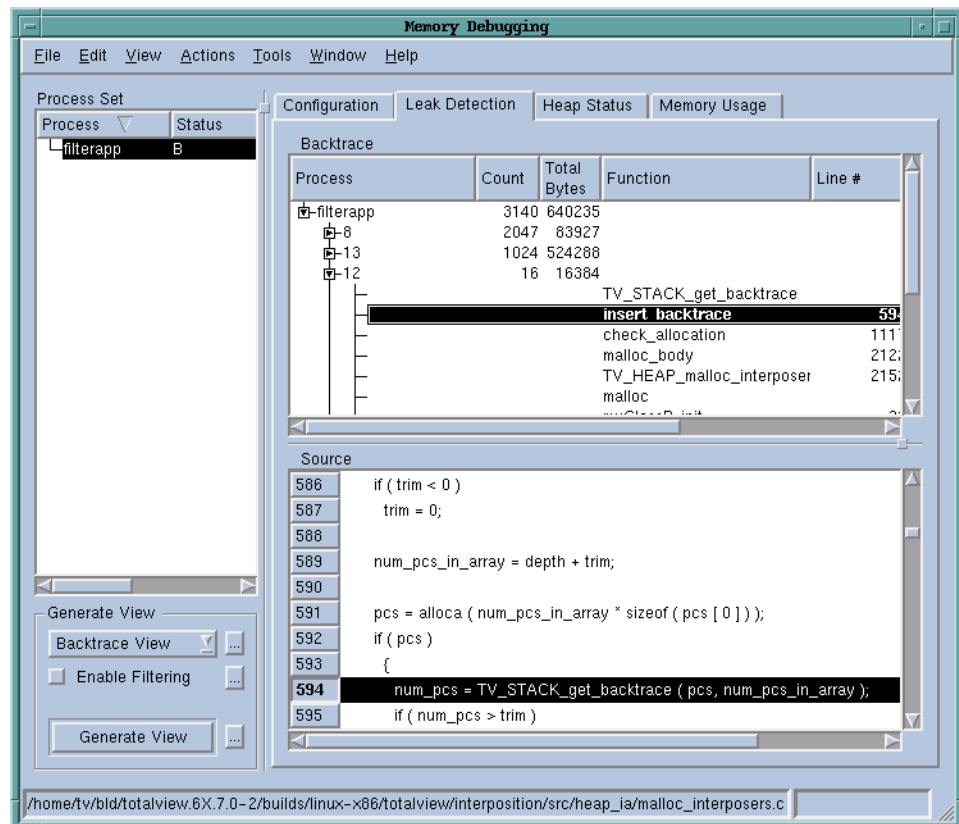
**Show byte counts as megabytes (MB) or kilobytes (KB)**

By default, the Memory Debugger displays memory sizes in KB. Selecting this check box tells the Memory Debugger to choose the most convenient size.

**Backtrace View**

The Backtrace View organizes the leaks in your program by the backtrace number created by the Memory Debugger. To create this view, select **Backtrace View**, and then select **Generate View**. In this view, the first column, **Process**, has a numeric list of all the backtrace ID numbers that the Memory Debugger creates.

Figure 52: Leak Detection  
Page: Backtrace View



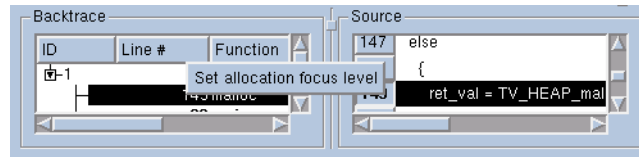
When you look at one backtrace, you might be seeing the rolling together of many leaks into one. You can tell how many leaks are associated with one ID by looking at the **Count** column. In this example, 16 leaks are associated with backtrace ID 12.

When you click on a line having a source code associated with it, the Memory Debugger displays that line in its Source Pane.

The backtrace being displayed is the one that existed when your program allocated the memory block. The Memory Debugger highlights the frame that it thinks is the one you should be focusing on. That is, it highlights where the memory allocation was made. If it guesses wrong, you can reset

the hierarchy of backtraces by right-clicking your mouse on the back trace that you want displayed, as follows.

Figure 53: Backtrace and Source Panes



From the context menu, select **Set allocation focus level**.

For example, assume that you have created a function named `my_malloc()` that filters all of your memory allocations. The Memory Debugger would probably guess that this is the function to highlight in the Backtrace Pane. However, you probably want to set the allocation focus on the function that called `my_malloc()`. Do this by selecting that function, and then right-clicking on it to invoke this command.

To set preferences associated with the Backtrace View, select the  button within the Generate View area on the left. The preferences are as follows:

### Check interior pointers during leak detection

Tells the Memory Debugger to consider a block as being referenced if a pointer is pointing anywhere within the block instead of just at the block's starting location. In most programs, the code should be keeping track of the block's boundary. However, if your C++ program is using multiple inheritance, you may be pointing into the middle of the block without knowing it.

Use this option with some caution as it can affect performance.

### Show byte counts as megabytes (MB) or kilobytes (KB)

By default, the Memory Debugger displays memory sizes in KB. Selecting this check box tells the Memory Debugger to choose the most convenient size.

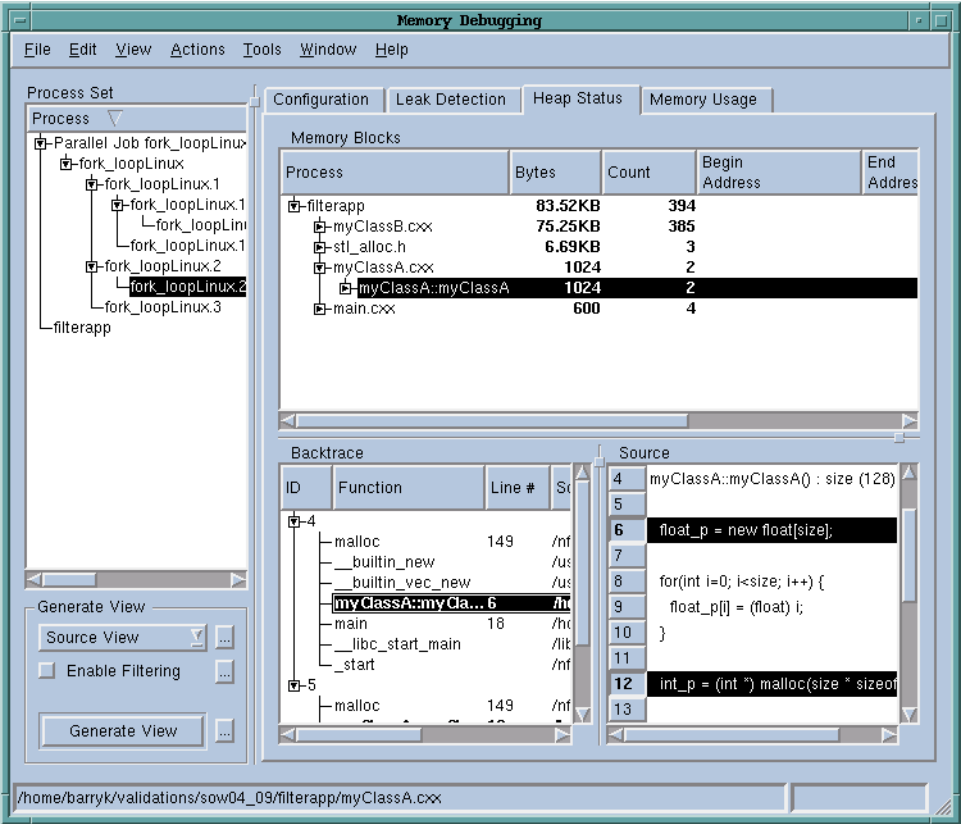
## Heap Status Page

The Heap Status Page displays information about all memory blocks that your program has not yet freed. The views shown in this page can be quite large. You can tell the Memory Debugger to display a Graphical, Source, or Backtrace View. Figure 54 on page 57 shows a Heap Status Source View.

### Source and Backtrace Views

The Source and Backtrace Views within the Leak Detection page contain the same type of information that these view contain with the Heap Status Page. The sole difference is, of course, that these views in the Heap Status Page contain all memory allocations, not just allocations that represent leaks.

Figure 54: Heap Status  
Page: Source View



In most cases, an individual item is not very remarkable or noteworthy. However, the “rolled-up” information about your allocations can help you better understand your program’s behavior.

For example, if your program’s size is greater than you’d expect it to be, you can select the **Bytes** column so that the largest allocations are all grouped together. Concentrating on the statements allocating the most memory should lead you understand your program’s behavior.

Similarly, if your program is allocating many small memory blocks, these allocations might be hurting performance. Looking at the information in the **Bytes** and **Count** columns might also give you some hints about where you can improve performance.

If you are displaying a Source View, you can display a **Block Properties** Window by right-clicking on a block in the top area, then selecting Properties. For more information, see “Memory Block Properties Window” on page 47.

You can also tell the Memory Debugger to display leaks in a different color. For more information, see “Heap Status Preferences” on page 59.

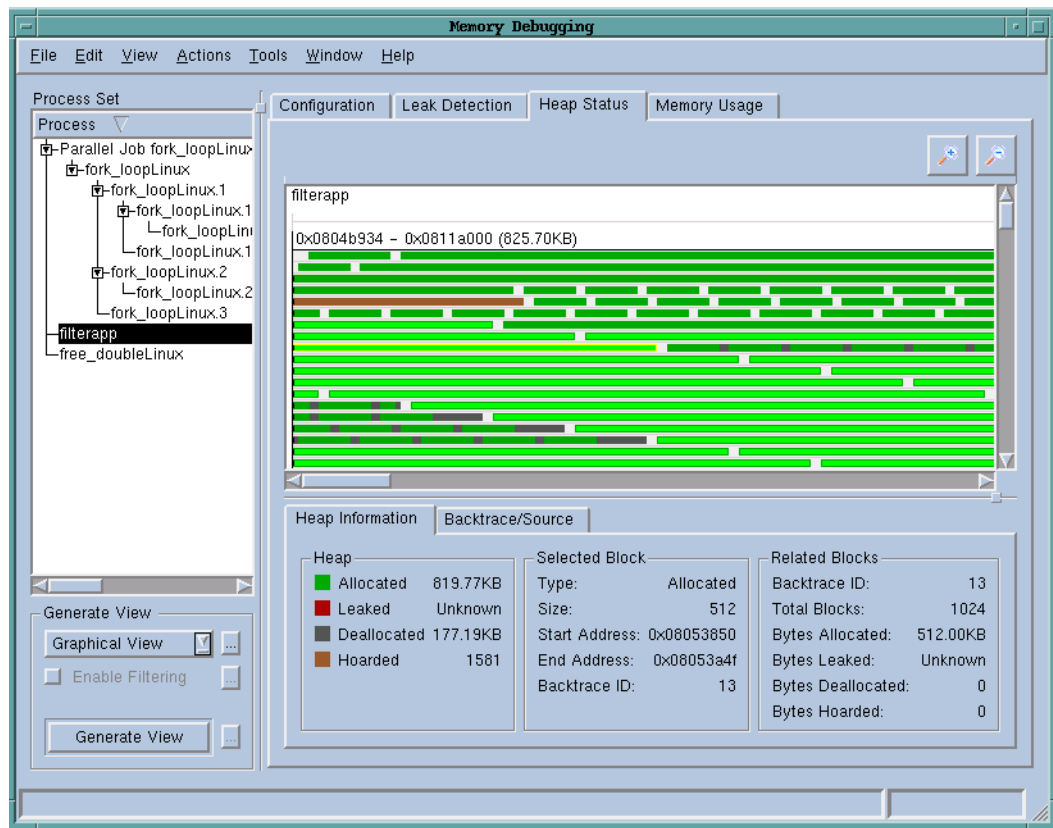
For more information on the contents of this page, see “Leak Detection Page” on page 52.

**Graphical View**

For programs making extensive use of the heap API, the information presented within the Heap Status views can be overwhelming. In these cases

and others, you may want to begin by displaying a graphical view of the heap. (See “Heap Status Page: Graphical View” on page 58)

Figure 55: Heap Status Page: Graphical View



You can create this view by selecting **Graphical View** and then pressing the **Generate View** button.

The Graphical View has two parts:

- The upper portion displays allocated blocks of memory.
- The bottom contains two tabs: **Heap Information** and **Backtrace/Source**. The information displayed when you select **Backtrace/Source** is the same as the Memory Debugger displays in the Source and Backtrace views. For information on the contents of these views, see “Leak Detection Page” on page 52

The length of each block in the upper portion is proportional to the size of the block. You can change the relative size of these blocks to see more or less information by selecting the magnifying glass icons above and to the right of the graphical display. The upper left corner within the graphical area contains general information.

The information in the top and bottom portions is linked. For example, if you select a block within the graphical area, the Memory Debugger displays information about the block in the bottom area. The Memory Debugger displays the selected block in yellow. It displays blocks having the same backtrace in green. If you are displaying the **Heap Information** Page, you’ll

see summary information about this block. If you are displaying the **Source/Backtrace** Page, you'll see the source line and backtrace associated with the block. If you select a source line or backtrace within this page, the Memory Debugger highlights the blocks associated with that source line and backtrace.

The three areas within the **Heap Information** page are as follows:

- **Heap:** Contains a key to the colors used in displaying blocks and a summary of how much memory is associated with each of the four allocation types displayed.
- **Selected Block:** Describes the block that you select. The only one of the five types whose meaning may be obscure is Backtrace ID. This is an identifier created by the Memory Debugger that it uses to associate different backtraces. You may find this number useful as you are examining memory information.
- **Related Blocks:** If the backtrace associated with a memory allocation is identical to the backtrace that existed when a previous allocation occurred, the Memory Debugger assigns the same backtrace ID to the newly created allocation. When you select a block, the Memory Debugger displays information about all blocks having the same backtrace ID.

You can display a **Block Properties** Window by right-clicking on a block in the top area, then selecting Properties. For more information, see "Memory Block Properties Window" on page 47.

## Heap Status Preferences


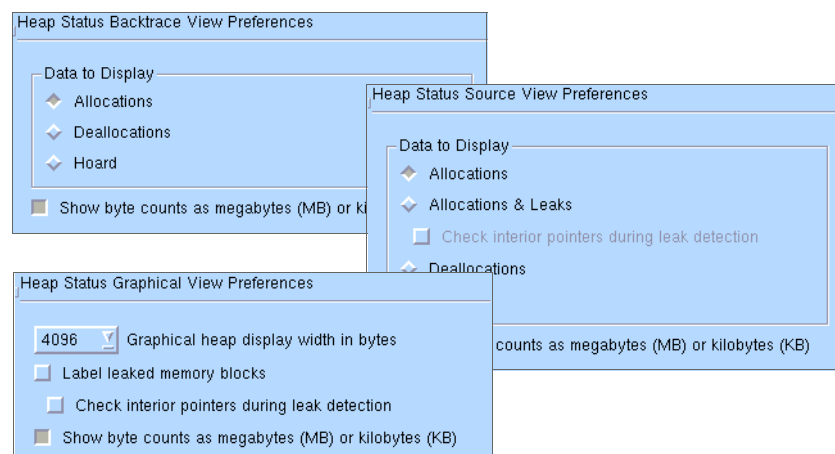
When you select the  button to the left of the view pulldown, the Memory Debugger displays a preference dialog box. The following figure shows the right side of each of the Heap Status preferences.

Figure 56: Heap Status Preferences:



Here is what these preferences let you do:

- Data to Display** (*Source and Backtrace View*) When displaying a Backtrace or Source View, tell the Memory Debugger to display allocations, deallocations, or hoarded information. In Source View, you can tell the Memory Debugger that it should also display leaked allocations.

### Label Leaked Memory

*(Graphical View)* Tells the Memory Debugger to display leaked memory in red.

### Check interior pointers during leak detection.

*(Source and Graphical View)* Tells the Memory Debugger to consider a block as being referenced if a pointer is pointing anywhere within the block instead of just at the block's starting location. In most programs, the code should be keeping track of the block's boundary. However, if your C++ program is using multiple inheritance, you may be pointing into the middle of the block without knowing it.

Use this option with some caution as it can affect performance.

### Graphical heap display width in bytes

*(Graphical View)* Defines how many bytes of block memory is displayed in each line within the graphical view. Don't confuse this with the zoom controls. The zoom controls increase and decrease the size the Memory Debugger uses to display blocks. That is, zooming just changes how much is visible at one time.

### Show byte counts as megabytes (MB) or kilobytes (KB)

*(all views)* When selected, the Memory Debugger chooses whether it should display memory in MB or KB. If this is not selected, the Memory Debugger always displays information in KB.

## Memory Usage Page

---

The Memory Usage Page tells you how your program is using memory, and where this memory is being used. One way to use this page is to compare memory use over time, so that you can tell if your program is leaking memory. If a program is leaking memory, you'll see that the amount of memory being used steadily increases over time. You can also compare memory use between processes, which can tell you if a process is using more memory than you expect.



*You do not need to enable memory debugging to obtain a Memory Usage View.*

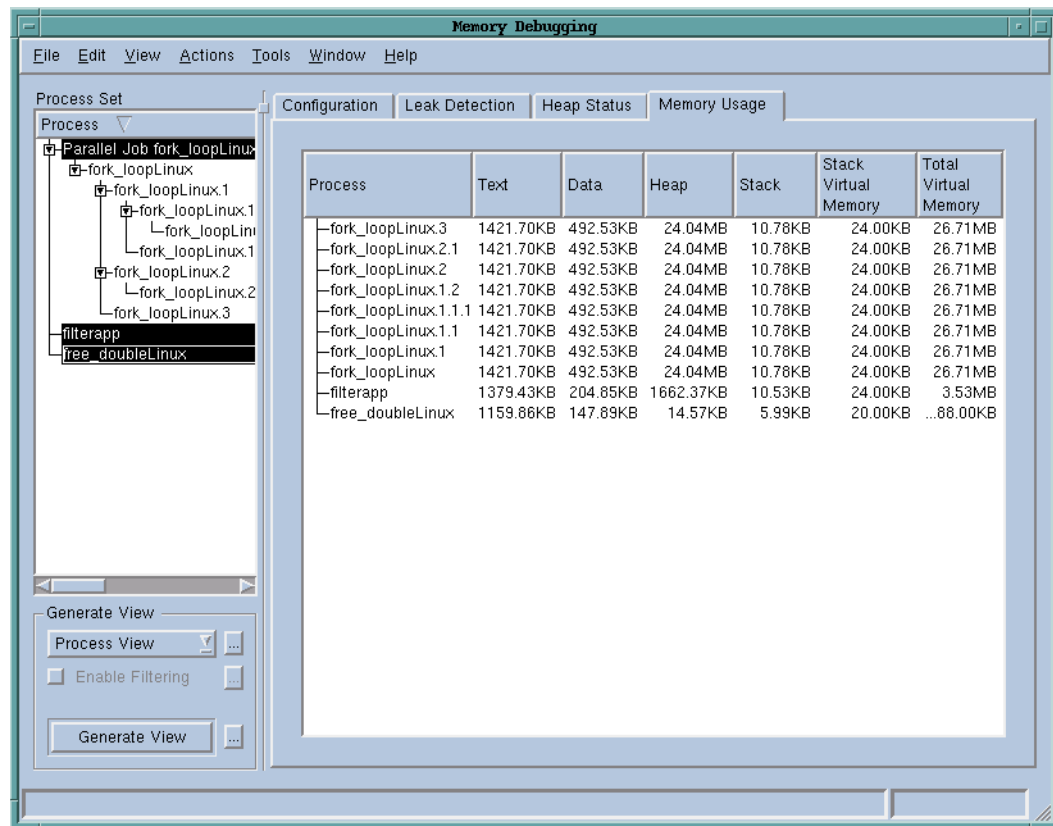
The Memory Debugger can present either a Process or Library View. The following figure shows an example of a Process View.

Clicking on a column header sorts the information from maximum to minimum, or vice versa.

If you add the memory values of all columns except the last, the sum doesn't equal the last column's value. There are several reasons for this. For example, most operating systems divide segments into pages, and information in a segment does not cross page boundaries. Another reason



Figure 57: Memory Usage Page: Process View



is that a process could map a file or an anonymous region. Areas such as these are part of what appear in the Stack Virtual Memory column. However, they do not appear elsewhere.

The information in these columns is as follows:

<b>Process</b>	The name of your process.
<b>Text</b>	The amount of memory used to store your program's machine code instructions.
<b>Data</b>	The amount of memory used to store uninitialized and initialized data.
<b>Heap</b>	The amount of memory currently being used for data created at run time.
<b>Stack</b>	The amount of memory used by the currently executing routine and all the routines in its backtrace.

If you are looking at a multi-threaded process, TotalView only shows information for the main thread's stack. The stack size of some threads does not change over time on some architectures.

On some systems, the space allocated for a thread is considered part of the heap.

**Stack Virtual Memory**

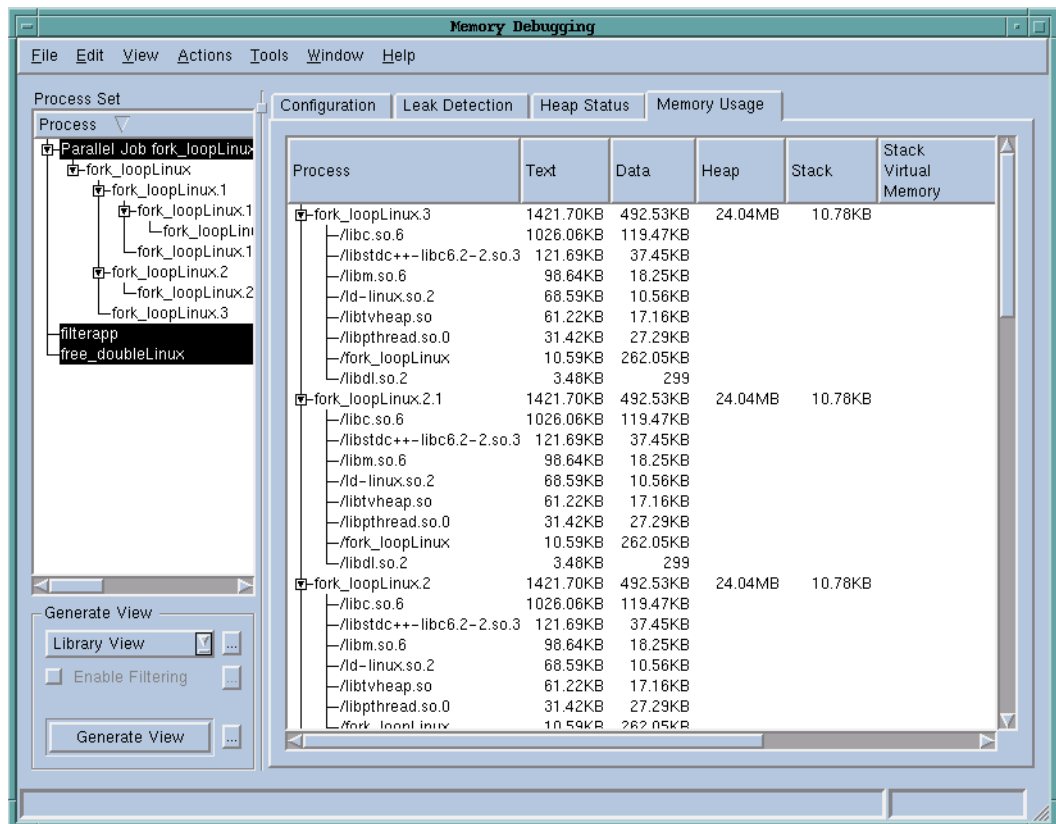
The logical size of the stack. This value is the difference between the current value of the stack pointer and the value reported in the **Stack** column. This value can differ from the size of the virtual memory mapping in which the stack resides.

**Total Virtual Memory**

The sum of the sizes of the mappings in the process's address space.

The Library Pane shows which library files are contained within your executable. In addition to the same kind of information shown in the Process View, this view shows the amount of memory used by the text and data segments of these libraries. (See the following figure.)

Figure 58: Memory Usage View: Library View



# Using the dheap Command

3

The **dheap** command lets you track memory problems from within the CLI. Although the **dheap** command lets you do everything that you can do using the GUI, there are also a few things that are unique to the CLI. The following list presents actions that you can perform in both:

- To see the status of the Memory Debugger, use the **dheap** command.
- To display information about the heap, use the **dheap -info** command. You can show information for the entire heap or limit what TotalView displays to just a part of it.
- To enable and disable the Memory Debugger, use the **dheap -enable** and **dheap -disable** commands.
- To start and stop error notification, use the **dheap -notify** and **dheap -nonotify** commands.
- To filter the information displayed, use the **dheap -filter** command.
- To check for leaks, use the **dheap -leaks** command.
- To paint memory with a bit pattern, use the **dheap -paint** command.
- To hoard memory, use the **dheap -hoard** command.



*There are several **dheap** options not yet available in the GUI.*

## dheap Example

The following example shows the kind of information that the CLI displays after the Memory Debugger locates an error:

```
d1.<> dheap
      process:  Enable  Notify  Available
      1      (18993):  yes    yes    yes
      1.1    realloc: Address does not match any allocated
      block.: 0xbfffd87c
```

```

dl.<> dheap -info -backtrace
process 1 (18993):
    0x8049e88 -- 0x8049e98 0x10 [ 16]
    flags: 0x0 (none)
    : realloc PC=0x400217e5 [../malloc_wrappers_dlopen.c]
    : argz_append PC=0x401ae025 [/lib/i686/libc.so.6]
    : __newlocale PC=0x4014b3c7 [/lib/i686/libc.so.6]
    :
...
../malloc_wrappers_dlopen.c]
    : main PC=0x080487c4 [../realloc_prob.c]
    : __libc_start_main PC=0x40140647 [/lib/i686/libc.so.6]
    : _start PC=0x08048621 [../realloc_prob]

    0x8049f18 -- 0x8049f3a 0x22 [ 34]
    flags: 0x0 (none)
    : realloc PC=0x400217e5 [../malloc_wrappers_dlopen.c]
    : main PC=0x0804883e [../realloc_prob.c]
    : __libc_start_main PC=0x40140647 [/lib/i686/libc.so.6]
    : _start PC=0x08048621 [../realloc_prob]

```

The information that is displayed in this example is explained in more detail later in this chapter.

## dheap

## Controls heap debugging

**Format:** Shows Memory Debugger state

**dheap [ -status ]**

Applies a saved configuration file

**dheap -apply\_config { default | filename }**

Shows information about a backtrace

**dheap -backtrace [ subcommands ]**

Enables or disables the Memory Debugger

**dheap { -enable | -disable }**

Enables or disables event notification

**dheap -event\_filter subcommands**

Writes memory information

**dheap -export subcommands**

Specifies which filters the Memory Debugger uses

**dheap -filter subcommands**

Enables or disables the retaining (hoarding) of freed memory blocks

**dheap -hoard [ subcommands ]**

Displays Memory Debugger information

**dheap -info [ subcommands ]**

Indicates whether an address is in a deallocated block

**dheap -is\_dangling address**

Locates memory leaks

**dheap -leaks [ -check\_interior ]**

Enables or disables Memory Debugger event notification

**dheap -[no]notify**

Paints memory with a distinct pattern

**dheap -paint [ subcommands ]**

Enables or disables allocation and reallocation notification

**dheap -tag\_alloc subcommand [ start\_address [ end\_address ] ]**

Displays the Memory Debugger version number

**dheap -version**

**Arguments:** [ -status ] Displays the current state of the Memory Debugger. This tells you if a process is capable of having its heap operations traced and if TotalView will notify you if a notifiable heap event occurs. If TotalView stops a thread because one of these events occur, it displays information about this event.

If you do not use other options to the **dheap** command, you can omit this option.

**-apply\_config** { **default** | *filename* }

Applies configuration settings within the named file to the Memory Debugger. If you type **default**, the Memory Debugger looks first in the current directory and then in your **.totalview/hia/** directory for a file named **default.hiarc**. Otherwise, it uses the name of the file you enter here. If you do not specify an extension, the Memory Debugger assumes that the extension is **.hiarc**. That is, while you can specify a file named **foo.foobar**, you cannot specify a file **foo** as the Memory Debugger would then assume that the file is actually named **foo.hiarc**.

**-backtrace** [ *subcommands* ]

Shows the current settings for the backtraces associated with a memory allocation. This information includes the *depth* and the *trim* (described later in this section).

**-status** Tells TotalView to display backtrace information. If you do not use other backtrace options, you can omit this option.

**-set\_depth** *depth*

**-reset\_depth**

Set or reset the *depth*. The *depth* is the maximum number of PCs that the Memory Debugger includes when it creates a backtrace. (The backtrace is created when a memory block is allocated or reallocated.) The *depth* value starts after the *trim* value. That is, the number of excluded frames does not include the trimmed frames.

When you use the **-reset\_depth** option, TotalView either restores its default setting or the setting you set using the **TV\_HEAP\_ARGS** environment variable.

**-set\_trim** *trim*

**-reset\_trim**

Sets or resets the *trim*. The *trim* describes the number of PCs from the top of the stack that the Memory Debugger ignores when it creates a backtrace. As the backtrace includes procedure calls from within the Memory Debugger, setting a trim value removes them from the backtrace. The default is to exclude Memory Debugger procedures. Similarly, your program might call the heap manager from within library code. If you do not want to see call frames showing a library, you can exclude them.

When you use the **-reset\_trim** option, TotalView either restores its default setting or the setting you set using the **TV\_HEAP\_ARGS** environment variable.

**-display** *backtrace\_id*

Displays the stack frames associated with the backtrace identified by *backtrace\_id*.

**-event\_filter** *subcommands*

The subcommands to this option let you control which agent events cause the Memory Debugger to stop program execution.

**-set { on | off }**

Enables or disables event filtering. If you disable event filtering, the Memory Debugger displays all events. If you enable event filtering, then you can control which events are displayed.

**-reset**

Resets the event filter to the Memory Debugger's default value. You can create your own default in a configuration file or by specifying an environment variable setting.

**-[no]notify** *event-list*

Enables or disables one or more events. The event names you can use are:

addr\_not\_at\_start  
alloc\_not\_in\_heap  
alloc\_null  
alloc\_returned\_bad\_alignment  
bad\_alignment\_argument  
dealloc\_notification  
double\_alloc  
free\_not\_allocated  
realloc\_not\_allocated  
realloc\_notification

**-export** *required\_subcmds* [ *optional\_subcmds* ]

Tells the Memory Debugger to write information to a file.

*required\_subcmds*

You must use all three of these options with **dheap**  
**-export**:

**-data { alloc | alloc\_leaks | dealloc | hoard | leaks }**

Specifies the data to be written into the exported file, as follows:

**alloc**: Show all heap allocations.

**alloc\_leaks**: Show all heap allocations and perform leak detection. This differs from the **alloc** argument in that TotalView annotations leaked allocations.

**dealloc**: Show deallocation data.

**hoard**: Show deallocations currently held in the hoard.

**leaks**: Show heap allocations that are leaked.

**-output** *filename*

Names the file into which TotalView writes memory information.

**-view { source | backtrace }**

Names the view to be exported.

*optional\_subcmds*

You can use any of the following options with **dheap**  
**-export**:

**-set\_show\_backtraces { on | off }**

When set to **on**, TotalView includes backtrace information within the data being written. As **on** is the default, you only need to use this option with the **off** argument.

**-set\_show\_code { on | off }**

When set to **on**, TotalView includes the source code for the place where the memory was allocated with the data being written. As **on** is the default, you only need to use this option with the **off** argument.

**-check\_interior**

Tells the Memory Debugger that a memory block should not be considered as leaked if a pointer is pointing anywhere within the block. TotalView ignores this option unless you also use the **-data\_leaks** option.

**-enable/-disable**

Using the **-enable** option tells TotalView to use the Memory Debugger agent to record heap events the next time you start the program. Using the **-disable** option tells TotalView to not use the agent the next time you start your program.

If necessary, you must preload the agent (see Chapter 4, "Creating Programs for Memory Debugging," on page 83 for information) before using this option.

**-filter subcommands**

Use the **-filter** options to enable, disable, and show information about filtering.

**-enable [ filter-name-list | all ]**

Enables filtering of **dheap** commands. If you do not use an argument with this option, this option is equivalent to selecting **Enable Filtering** in the Memory Debugger Window.

If you use a filter name, you are telling the Memory Debugger where to locate filter information. You still need to enable filtering. For example, here is how you would enable filtering and enable the use of a filter named **MyFilter**:

```
dheap -filter -enable MyFilter
```

```
dheap -filter -enable
```

If you did not enter the second command, no filtering occurs.

The **all** argument tells the Memory Debugger to enable all of your filters.

**-disable [ filter-name-list | all ]**

Disables filtering or disables an individual filter. The way that you use this command is similar to **dheap -filter -enable**.



**-list** [ **[-full]** *filter-name-list* ]

Displays a filter description and its enabled state. If you do not use a *filter-name* argument, the CLI displays all defined filters and their enabled states.

If you include the full argument, the information includes all of the filter's criteria.

**-hoard** [ *subcommands* ]

Tells the Memory Debugger not to surrender allocated blocks back to your program's heap manager. If you do not type a subcommand, the Memory Debugger displays information about the hoarded blocks. For more information, see "Memory Reuse: *dheap -hoard*" on page 75.

[ **-status** ] Displays hoard settings. Information displayed indicates if hoarding is enabled, if deallocated blocks are added to the hoard (or only those that are tagged), the maximum size of the hoard, and the hoard's current size.

If you do not use other hoarding options, you can omit the **-status** option when you want to see status information.

**-display** [ *start\_address* [ *end\_address* ] ]

Displays the contents of the hoard. You can restrict the display by specifying *start\_address* and *end\_address*. If you omit *end\_address* or use a value of 0, the Memory Debugger displays all contents beginning at *start\_address* and going to the end of the hoard.

The CLI displays hoarded blocks in the order in which your program deallocated them.

**-set** [ **on** | **off** ]

Enables and disables hoarding.

**-reset**

Resets the Memory Debugger settings for hoarding back to their initial value.

**-set\_all\_deallocs** [ **on** | **off** ]

Tells the Memory Debugger whether to hoard deallocated blocks.

**-reset\_all\_deallocs**

Resets the Memory Debugger settings for hoarding of deallocated blocks to its initial value.

**-set\_max\_kb** *num\_kb*

Sets the maximum size of the hoarded information.

**-set\_max\_blocks** *num\_blocks*

Set the maximum number of hoarded blocks.

**-reset\_max\_kb**

**-reset\_max\_blocks**

Resets a hoarding size value back to its default.

**-info [ *subcommand* ]**

Displays information about the heap or regions of the heap within a range of addresses. If you do not use the address arguments, the CLI displays information about all heap allocations.

The information that the Memory Debugger displays includes the start address, a block's length, and other information such as flags or attributes.

**-backtrace**

Tells the CLI to display backtrace information. This list can be very long.

*start\_address*

If you just type a *start\_address*, the CLI reports on all allocations beginning at and following this address. If you also type an *end\_address*, the CLI limits the display to those allocations between the *start\_address* and the *end\_address*.

*end\_address*

If you also specify an *end\_address*, the CLI reports on all allocations between *start\_address* and *end\_address*. If you type 0, it's the same as omitting this argument; that is, the Memory Debugger displays information from the *start\_address* to the end of the address space.

**-is\_dangling *address***

Indicates if an *address* that was once allocated and not yet recycled by the heap manager is now deallocated.

**-leaks**

Locates all memory blocks that your program allocated and which are no longer referenced. That is, using this command tells the Memory Debugger to locate all dangling memory. For more information, see "Detecting Leaks: *dheap -leaks*" on page 79.

By default, the Memory Debugger only checks to see if the starting location of an allocated memory block is referenced.

**-check\_interior**

Tells the Memory Debugger to consider a memory block as being referenced if the interior portion of it is referenced.

**-[no]notify**

Using the **-notify** option tells TotalView to stop your program's execution when the Memory Debugger detects a notifiable event, and then print a message (or display a dialog box if you are also using the GUI) that explains what just occurred. The Memory Debugger can notify you when heap memory errors occur or when tagged blocks are deallocated or reallocated.

Using the **-nonotify** option tells TotalView not to stop execution. Even if you type the **-nonotify** option, TotalView tracks heap events.

**-paint** [ *subcommands* ]

Enables and disables block painting and shows status information. (For more information on block painting, see “Block Painting: *dheap -paint*” on page 79.)

[ **-status** ] Shows the current paint settings. These are either the values you set using other painting options or their default values.

If you do not use a subcommand to the **-paint** option, the Memory Debugger shows the block painting status information.

**-set\_alloc** [ *on* | *off* ]

**-set\_dealloc** [ *on* | *off* ]

**-set\_zalloc** [ *on* | *off* ]

The *on* options enable block painting. They tell the Memory Debugger to paint a block when your program’s heap manager allocates, deallocates, or uses a memory function that sets memory blocks to zero.

You can only paint zero-allocated blocks if you are also painting regular allocations.

The *off* options disable block painting.

**-reset\_alloc**

**-reset\_dealloc**

**-reset\_zalloc**

Reset the Memory Debugger settings for block painting to their initial values or to values typed in a startup file.

**-set\_alloc\_pattern** *pattern*

**-set\_dealloc\_pattern** *pattern*

Set the pattern that the Memory Debugger uses the next time it paints a block of memory. The maximum width of *pattern* can differ between operating systems. However, your pattern can be shorter.

**-reset\_alloc\_pattern**

**-reset\_dealloc\_pattern**

Reset the patterns used when the Memory Debugger paints memory to the Memory Debugger default values.

**-tag\_alloc** *subcommand* [ *start\_address* [ *end\_address* ] ]

Tells the Memory Debugger to mark a block so that it can notify you when your program deallocates or reallocates a memory block. (For more information, see “Deallocation Notification: *dheap -tag\_alloc*” on page 80.)

When tagging memory, if you do not specify address arguments, the Memory Debugger either tags all allocated blocks or removes the tag from all tagged blocks.

**-[no]hoard\_on\_dealloc**

Tells the Memory Debugger that it should not release tagged memory back to your program’s heap manager for reuse when it is deallocated—this is used in conjunction with hoarding. To reenable memory reuse, use

the **-nohoard\_on\_dealloc** subcommand. See “Memory Reuse: *dheap -hoard*” on page 75 for more information.

If you use this option, the memory tracker only hoards tagged blocks. In contrast, if you use the **dheap -hoard -set\_all\_deallocs on** command, the Memory Debugger hoards all deallocated blocks.

**-[no]notify\_dealloc**

**-[no]notify\_realloc**

Enable or disable notification when your program deallocates or reallocates a memory block.

*start\_address*

If you only type a *start\_address*, the Memory Debugger either tags or removes the tag from the block that contains this address. The action it performs depends on the subcommand you use.

*end\_address*

If you also specify an *end\_address*, the Memory Debugger either tags all blocks beginning with the block containing the *start\_address* and ending with the block containing the *end\_address* or removes the tag. The action it performs depends on the subcommand you use. If *end\_address* is 0 (zero) or you do not type an *end\_address*, the Memory Debugger tags or removes the tag from all addresses beginning with *start\_address* to the end of the heap.

**-version**

Displays the Memory Debugger version number.

**Description:** The **dheap** command controls the TotalView Memory Debugger. The Memory Debugger can:

- Tell TotalView to use the Memory Debugger agent to track memory errors.
- Stop execution when a **free()** error occurs, and display information you need to analyze the error. For more information, see “Notification When free Problems Occur” on page 74.
- Hoard freed memory so that it is not released to the heap manager. For more information, see “Memory Reuse: *dheap -hoard*” on page 75.
- Write heap information to a file. For more information, see “Writing Heap Information: *dheap -export*” on page 77.
- Remove unwanted information from displays. For more information, see “Filtering Heap Information: *dheap -filter*” on page 77.
- Detect leaked memory by analyzing if a memory block is reachable. For more information, see “Detecting Leaks: *dheap -leaks*” on page 79.
- Paint memory with a bit pattern when it is allocated and deallocated. For more information, see “Block Painting: *dheap -paint*” on page 79.
- Notify you when a memory block is deallocated or reallocated. For more information, see “Deallocation Notification: *dheap -tag\_alloc*” on page 80.

The first step when debugging memory problems is to type the **dheap** **-enable** command. This command activates the Memory Debugger. You must do this before your program begins executing. If you try to do this after execution starts, TotalView tells you that it will enable the Memory Debugger when you restart your program. For example:

```
d1.<> n
    64 >  int          num_reds      = 15;
d1.<> dheap -enable
process 1 (30100): This will only take effect on restart
```

You can tell the Memory Debugger to stop execution if:

- A **free()** problem exists by using the **dheap -notify** command.
- A block is deallocated by using the **dheap -tag\_alloc -notify\_dealloc** command.
- A block is reallocated by using the **dheap -tag\_alloc -notify\_realloc** command.

If you enable notification, TotalView stops the process when it detects one of these events. The Memory Debugger is always monitoring heap events, even if you turned notification off. That is, disabling notification means that TotalView does not stop a program when events occur. In addition, it does not tell you that the event occurred.

While you can separately enable and disable notification in any group, process, or thread, you probably only want to activate notification on the control group's master process. Because this is the only process that TotalView creates, it is the only process where TotalView can control the Memory Debugger's environment variable. For example, slave processes are normally created by an MPI starter process or as a result of using the **fork()** and **exec()** functions. In these cases, TotalView simply attaches to them. For more information, see Chapter 4, "Creating Programs for Memory Debugging," on page 83.

If you do not use a **dheap** subcommand, the CLI displays memory status information. You only use the **-status** option when you want the CLI to display status information in addition to doing something else.

The information that the **dheap** command displays can contain a flag containing additional information about the memory location. The following table describes these flags:

Flag Value	Meaning
0x0001	Operation in progress
0x0002	<b>notify_dealloc</b> : you will be notified if the block is deallocated
0x0004	<b>notify_realloc</b> : you will be notified if the block is reallocated
0x0008	<b>paint_on_dealloc</b> : the Memory Debugger will paint the block when it is deallocated
0x0010	<b>dont_free_on_dealloc</b> : the Memory Debugger will not free the tagged block when it is deallocated
0x0020	<b>hoarded</b> : the Memory Debugger is hoarding the block

While some **dheap** options obtain information on specific memory conditions, you can use the following options throughout your session:

- **dheap** or **dheap -status**: Displays Memory Debugger state information. For example:

```
a1.<> dheap -status
      process:      Enable    Notify    Available
      1      (18868):      yes      yes      yes
      2      (18947):      n/a      yes      yes
      3      (18953):      n/a      yes      yes
      4      (18956):      n/a      yes      yes
```

- **dheap -version**: Displays version information. You receive information for each process as processes can be compiled with different versions of the Memory Debugger. For example:

```
a1.<> dheap -version
      process:      Version
      1      (18868):      1.001
      2      (18947):      1.001
      3      (18953):      1.001
      4      (18956):      1.001
```

- **dheap -backtrace**: Displays information about how much of the backtrace is being displayed. For example:

```
a1.<> dheap -backtrace
      process:      Depth      Trim
      1      (18868):      32      5
      2      (18947):      32      5
      3      (18953):      32      5
      4      (18956):      32      5
```

Using arguments to this command, you can change both the *depth* and the *trim* values. Changing the *depth* value changes the number of stack frames that the Memory Debugger displays in a backtrace display. Changing the *trim* value eliminates the topmost stack frames.

- **dheap -info**: Displays information about currently allocated memory blocks. For example:

```
d1.<> dheap -info
process 1 (5320):
      0x8049790 --      0x804979a      0xa [      10]
      flags: 0x0 (none)
      0x80497a0 --      0x80497b4      0x14 [      20]
      flags: 0x0 (none)
      0x80497b8 --      0x80497d6      0x1e [      30]
      flags: 0x0 (none)
      0x80497e0 --      0x8049808      0x28 [      40]
      flags: 0x0 (none)
```

### Notification When free Problems Occur

If you type **dheap -enable -notify** and then run your program, the Memory Debugger notifies you if a problem occurs when your program tries to free memory. (For more information, see Chapter 15 of the *TotalView Users Guide*.)

When execution stops, you can type **dheap** (with no arguments), to show information about what happened. You can also use the **dheap -info** and **dheap -info -backtrace** commands to display additional information. The

information displayed by these commands lets you locate the statement in your program that caused the problem. For example:

```
d1.<> dheap
      process:  Enable   Notify   Available
1      (18993):   yes     yes      yes
1.1    realloc: Address does not match any allocated block.:
          0xbfffd87c
```

For each allocated region, the CLI displays the start and end address, and the length of the region in decimal and hexadecimal formats. For example:

```
d1.<> dheap
      process:  Enable   Notify   Available
1      (30420):   yes     yes      yes
1.1    free: Address is not the start of any allocated block.:
      free: existing allocated block:
      free: start=0x08049b00 length=(17 [0x11])
      free: flags: 0x0 (none)
      free: malloc      PC=0x40021739 [./.../
malloc_wrappers_dlopen.c]
      free: main        PC=0x0804871b [./free_prob.c]
      free: __libc_start_main PC=0x40140647 [./lib/i686/
libc.so.6]
      free: _start      PC=0x080485e1 [./.../free_prob]

      free: address passed to heap manager: 0x08049b08
```

The Memory Debugger can also tell you when tagged blocks are deallocated or reallocated. For more information, see “*Deallocation Notification: dheap -tag\_alloc*” on page 80.

### Showing Backtrace Information: dheap -backtrace:

The backtrace associated with a memory allocation can contain many stack frames that are part of the heap library, the Memory Debugger’s library, and other related functions and libraries. You are not usually interested in this information, since these stack frames aren’t part of your program. Using the **-backtrace** option lets you manage this information, as follows:

#### ■ dheap -backtrace -set\_trim *value*

Tells the Memory Debugger to remove—that is, trim—this number of stack frames from the top of the backtrace. This lets you *hide* the stack frames that you’re not interested in as they come from libraries.

#### ■ dheap -backtrace -set\_depth *value*

Tells the Memory Debugger to limit the number of stack frames to the value that you type as an argument. The *depth* value starts after the *trim* value. That is, the number of excluded frames does not include the frames that were trimmed.

### Memory Reuse: dheap -hoard

In some cases, you may not want your system’s heap manager to immediately reuse memory. You would do this, for example, when you are trying to find problems that occur when more than one process or thread is allocating the same memory block. Hoarding allows you to temporarily delay the

block's release to the heap manager. When the hoard has reached its capacity in either size or number of blocks, the Memory Debugger releases previously hoarded blocks back to your program's heap manager.

The order in which the Memory Debugger releases blocks is the order in which it hoards them. That is, the first blocks hoarded are the first blocks released—this is a first-in, first-out (fifo) queue.

Hoarding is a two-step process, as follows:

- 1 Use the **dheap -enable** command to tell the Memory Debugger to track heap allocations.
- 2 Use the **dheap -hoard -set on** command to tell the Memory Debugger not to release deallocated blocks back to the heap manager. (The **dheap -hoard -set off** command tells the Memory Debugger to no longer hoard memory.) After you turn hoarding on, use the **dheap -hoard -set\_all\_deallocs on** command to tell the Memory Debugger to start hoarding blocks.

At any time, you can obtain the hoard's status by typing the **dheap -hoard** command. For example:

```
d1.<> dheap -hoard
```

		All	Max	Max		
process:	Enabled	deallocs	size	blocks	Size	Blocks
1 (10883):	yes	yes	16 (kb)	32	15 (kb)	9

The **Enabled** column contains either **yes** or **no**, which indicates whether hoarding is enabled. The **All deallocs** column indicates if hoarding is occurring. The next columns show the maximum size in kilobytes and number of blocks to which the hoard can grow. The last two columns show the current size of the hoard, again, in kilobytes and the number of blocks.

As your program executes, the Memory Debugger adds the deallocated region to a FIFO buffer. Depending on your program's use of the heap, the hoard could become quite large. You can control the hoard's size by setting the maximum amount of memory in kilobytes that the Memory Debugger can hoard and the maximum number of hoarded blocks.

**dheap -hoard -set\_max\_kb num\_kb**

Sets the maximum size in kilobytes to which the hoard is allowed to grow. The default value on many operating systems is 32KB.

**dheap -hoard -set\_max\_blocks num\_blocks**

Sets the maximum number of blocks that the hoard can contain.

You can tell which blocks are in the hoard by typing the **dheap -hoard -display** command. For example:

```
d1.<> dheap -hoard -display
```

process	1	(10883):			
	0x804cdb0	--	0x804d3b0	0x600	[ 1536]
flags:	0x32 (hoarded)				



```

0x804d3b8 -- 0x804dab8 0x700 [ 1792]
flags: 0x32 (hoarded)
0x804dac0 -- 0x804e2c0 0x800 [ 2048]
flags: 0x32 (hoarded)
0x804fce8 -- 0x804fee8 0x200 [ 512]
flags: 0x32 (hoarded)
0x804fef0 -- 0x80502f0 0x400 [ 1024]
flags: 0x32 (hoarded)

```

### Writing Heap Information: dheap -export

You may want to write the information that the Memory Debugger collects about your program to disk so that you can examine it at a later time. Or, you may want to save information from different sessions so that you can compare changes that you've made.

You can save Memory Debugger information by using the **dheap -export** command. This command has two sets of options: one contain options you must specify, the other contains options that are optional. In all cases, you must use the:

- **-output** option to name the file to which the Memory Debugger writes information.
- **-view** option to indicate if you want either a source or backtrace view.
- **-data** option to name which data is included.

For example:

```
dheap -export -output heap.txt -view source -data leaks
```

You can also add **-set\_show\_code** and **-set\_show\_backtraces**. These options are most often used to restrict the amount of information being displayed. You can also use the **-check\_interior** option to tell the Memory Debugger that if a pointer is pointing into a block instead of at the block's beginning, then the block shouldn't be considered as being leaked.

### Filtering Heap Information: dheap -filter

Depending upon the way in which your program manages memory, the Memory Debugger might be managing a lot of information. You can filter this information down to focus on things that are important to you at the moment by using filters. These filters can only be created using the GUI. However, after you create a filter using the GUI, you can apply it from within the CLI by using the **dheap -filter** commands.

Here is an excerpt from a CLI interaction:

```

dl.<> dheap -filter -list
Filtering of heap reports is 'disabled'
Individual filters are set as follows:
    Disabled  MyFilter  Function  contains  strdup

dl.<> dheap -filter -enable MyFilter
dl.<> dheap -filter -enable
dl.<> dheap -filter -list
Filtering of heap reports is 'enabled'
Individual filters are set as follows:

```

```
Enabled MyFilter Function contains strdup
```

```
d1.<>
```

Notice that TotalView automatically knew about your filters. That is, it always reads your filter file. However, TotalView ignores the file until you both enable the file and enable filtering. That is, while the following two commands look about the same, they are different:

```
dheap -filter -enable MyFilter
dheap -filter -enable
```

The first command tells the Memory Debugger that it could use the information contained within the **MyFilter** filter. However, the Memory Debugger only uses it after you enter the second command.

### Checking for Dangling Pointers: **dheap -is\_dangling:**

The **dheap -is\_dangling** command lets you determine if a pointer is still pointing into a deallocated memory block.

You can also use the **dheap -is\_dangling** command to determine if an address refers to a block that was once allocated but has not yet been recycled. That is, this command lets you know if a pointer is pointing into deallocated memory.

Here's a small program that illustrates a dangling pointer:

```
main(int argc, char **argv)
{
    int *addr = 0;          /* Pointer to start of block. */
    int *misaddr = 0;       /* Pointer to interior of block. */

    addr = (int *) malloc (10 * sizeof(int));
                          /* Point to interior of the block. */
    misaddr = addr + 5;

                          /* addr and misaddr now dangling. */
    free (addr);
    printf ("addr=%lx, misaddr=%lx\n",
           (long) addr, (long) misaddr);
}
```

If you set a breakpoint on the **printf()** statement and probe the addresses of **addr** and **misaddr**, the CLI displays the following:

```
d1.<> dheap -is_dangling 0x80496d0
      process:          0x80496d0
      1      (19405):    dangling

d1.<> dheap -is_dangling 0x80496e4
      process:          0x80496e4
      1      (19405):    dangling interior
```

This example is contrived. When creating this example, the variables were examined for their address and their addresses were used as arguments. In a realistic program, you'd find the memory block referenced by a pointer and then use that value. In this case, because it is so simple, using the CLI **dprint** command gives you the information you need. For example:

```

dl.<> dprint addr
addr = 0x080496d0 (Dangling) -> 0x00000000 (0)
dl.<> dprint misaddr
misaddr = 0x080496e4 (Dangling Interior) -> 0x00000000 (0)

```

If a pointer is pointing into memory that is deallocated, and this memory is being hoarded, the CLI also lets you know that you are looking at hoarded memory.

### Detecting Leaks: dheap -leaks

The **dheap -leaks** command locates memory blocks that were allocated and are no longer referenced. It then displays a report that describes these blocks; for example:

```

dl.<> dheap -leaks
process 1 (32188): total count 9, total bytes 450
* leak 1 -- total count 9 (100.00%), total bytes 450 (100%)
-- smallest / largest / average leak: 10 / 90 / 50
: malloc PC=0x40021739 [../malloc_wrappers_dlopn.c]
: main PC=0x0804851e [../local_leak.cxx]
: __libc_start_main PC=0x40055647 [/lib/i686/libc.so.6]
: _start PC=0x080483f1 [../local_leak]

```

If you use the **-check\_interior** option, the Memory Debugger considers a block as being referenced if a pointer exists to memory inside the block.

In addition to providing backtrace information, the CLI:

- Consolidates leaks made by one program statement into one leak report. For example, leak 1 has nine instances.
- Reports the amount of memory consumed for a group of leaks. It also tells you what percentage of leaked memory this one group of memory is using.
- Indicates the smallest and largest leak size, as well as telling you what the average leak size is for a group.

You might want to paint a memory block when it is deallocated so that you can recognize that the data pointed to is out-of-date. Tagging the block so that you can be notified when it is deallocated is another way to locate the source of problems.

### Block Painting: dheap -paint

When your program allocates or deallocates a block, the Memory Debugger can paint the block with a bit pattern. This makes it easy to identify uninitialized blocks, or blocks pointed to by dangling pointers.

Here are the commands that enable block painting:

- **dheap -paint -set\_alloc on**
- **dheap -paint -set\_dealloc on**
- **dheap -paint -set\_zalloc on**

Use the **dheap -paint** command to check the kind of painting that occurs and what the current painting pattern is. For example:

```
d1.<> dheap -paint

           process: Alloc Dealloc AllocZero   Alloc      Dealloc
           1      (1012): yes      yes      no  0xa110ca7f  0xdea110cf
```

Some heap allocation routines such as `calloc()` return memory initialized to zero. Using the `-set_zalloc_on` command allows you to separately enable the painting of the memory blocks altered by these kinds of routines. If you do enable painting for routines that set memory to zero, the Memory Debugger uses the same pattern that it uses for a normal allocation.

Here's an example of painted memory:

```
d1.<> dprint *(red_balls)
*(red_balls) = {
    value = 0xa110ca7f (-1592735105)
    x = -2.05181867705792e-149
    y = -2.05181867705792e-149
    spare = 0xa110ca7f (-1592735105)
    colour = 0xa110ca7f -> <Bad address: 0xa110ca7f>
}
```

The `0xa110ca7f` allocation pattern resembles the word "allocate". Similarly, the `0xdea110cf` deallocation pattern resembles "deallocate".

Notice that all of the values in the `red_balls` structure in this example aren't set to `0xa110ca7f`. This is because the amount of memory used by elements of the variable use more bits than the `0xa110ca7f` bit pattern. The following two CLI statements show the result of printing the `x` variable, and then casting it into an array of two integers:

```
d1.<> dprint (red_balls)->x
(red_balls)->x = -2.05181867705792e-149
d1.<> dprint {*(int[2]*)&(red_balls)->x}
*(int[2]*)&(red_balls)->x = {
    [0] = 0xa110ca7f (-1592735105)
    [1] = 0xa110ca7f (-1592735105)
```

(Diving in the GUI is much easier.)

You can tell the Memory Debugger to use a different pattern by using the following two commands:

- `dheap -paint -set_alloc_pattern pattern`
- `dheap -paint -set_dealloc_pattern pattern`

### Deallocation Notification: `dheap -tag_alloc`

You can tell the Memory Debugger to tag information within the Memory Debugger's tables and to notify you when your program either frees a block or passes it to `realloc()` by using the following two commands:

- `dheap -tag_alloc -notify_dealloc`
- `dheap -tag_alloc -notify_realloc`

Tagging is done within the Memory Debugger's agent. It tells the Memory Debugger to watch those memory blocks. Arguments to these commands tell the Memory Debugger which blocks to tag. If you do not type address arguments, TotalView notifies you when your program frees or reallocates

an allocated block. The following example shows how to tag a block and how to see that a block is tagged:

```
d1.<> dheap -tag_alloc -notify_dealloc 0x8049a48
process 1 (19387): 1 record(s) update
d1.<> dheap -info
process 1 (19387):
    0x8049a48 --          0x8049b48      0x100 [      256]
    flags: 0x2 (notify_dealloc)
    0x8049b50 --          0x8049d50      0x200 [      512]
    flags: 0x0 (none)
    0x8049d58 --          0x804a058      0x300 [      768]
    flags: 0x0 (none)
```

Using the **-notify\_dealloc** subcommand tells the Memory Debugger to let you know when a memory block is freed or when **realloc()** is called with its length set to zero. If you want notification when other values are passed to the **realloc()** function, use the **-notify\_realloc** subcommand.

After execution stops, here is what the CLI displays when you type another **dheap -info** command:

```
d1.<> dheap -info
process 1 (19387):
    0x8049a48 --          0x8049b48      0x100 [      256]
    flags: 0x3 (notify_dealloc, op_in_progress)
    0x8049b50 --          0x8049d50      0x200 [      512]
    flags: 0x0 (none)
    0x8049d58 --          0x804a058      0x300 [      768]
```

## TV\_HEAP\_ARGS

Environment variable for presetting Memory Debugger values

When you start TotalView, it looks for the **TV\_HEAP\_ARGS** environment variable. If it exists, TotalView reads values placed in it. If one of these values changes a Memory Debugger default value, the Memory Debugger uses this value as the default.

If you select a **<Default>** button in the GUI or a reset option in the CLI, the Memory Debugger resets the value to the one you set here, rather than to its default.

### TV\_HEAP\_ARGS Values

The values that you can enter into this variable are as follows:

#### **display\_allocations\_on\_exit**

Tells the Memory Debugger to dump the allocation table when your program exits. If your program ends because it received a signal, the Memory Debugger might not be able to dump this table.

#### **backtrace\_depth** *depth*

Sets the backtrace depth value. See “*Showing Backtrace Information: dheap –backtrace:*” on page 75 for more information.

#### **backtrace\_trim** *trim*

Sets the backtrace trim value. See “*Showing Backtrace Information: dheap –backtrace:*” on page 75 for more information.

#### **memalign\_strict\_alignment\_even\_multiple**

The Memory Debugger provides an integral multiple of the alignment rather than the even multiple described in the Sun **memalign** documentation. By including this value, you are telling the Memory Debugger to use the Sun alignment definition. However, your results might be inconsistent if you do this.

#### **output fd** *int*

#### **output file** *pathname*

Sends output from the Memory Debugger to the file descriptor or file that you name.

#### **verbosity** *int*

Sets the Memory Debugger’s verbosity level. If the level is greater than 0, the Memory Debugger sends information to **stderr**. The values you can set are:

0: Display no information. This is the default.

1: Print error messages.

2: Print all relevant information.

This option is most often used when debugging Memory Debugger problems. Setting the TotalView **VERBOSE** CLI variable does about the same thing.

### Example:

When you are entering more than one value, separate entries with spaces. For example:

```
setenv TV_HEAP_ARGS output file “my_file backtrace_depth 16”
```

# Creating Programs for Memory Debugging

## 4

The TotalView Memory Debugger puts its heap agent between your program and its heap library. This allows the agent to intercept the calls that your program makes to this library. After it intercepts the call, it checks it for errors, and then sends it on to the library so that it can be processed. The Memory Debugger agent does not replace standard memory functions; it just monitors what they do. For more information, see “*Behind the Scenes*” on page 5.

You can incorporate the agent into your environment either by:

- Linking your application with the agent.
- Requesting that the agent’s library be preloaded by setting a run-time loader environment variable. This is only done when your program will attach to another program that it did not start and you want the Memory Debugger to locate problems in this second application.

AIX applications differ from applications running on other platforms as AIX does not support interposition. However, TotalView can replace the AIX heap library.

Topics in this chapter are:

- “*Linking Your Application With the Agent*” on page 83
- “*Attaching to Programs*” on page 85
- “*Using the Memory Debugger*” on page 86
- “*Installing tvheap\_mr.a on AIX*” on page 88

## Linking Your Application With the Agent

In some situations, you need to explicitly link the Memory Debugger’s agent directly to your program. For example, if you are debugging an MPI program, your starter program might not propagate environment variables.



On AIX, you must always link your program so that `malloc()` can find the heap replacement and agent. In addition, you only set your `LIBPATH` environment variable when the `tvheap_mr.a` library is in your `LIBPATH`. If it isn't, your program might not load. You must use the `-L` options listed in the following table.

The following table lists additional linker command-line options that you must use when you link your program:

Platform	Compiler	ABI	Additional linker options
HP Tru64 Alpha (version 5)	Compaq/KCC	64	<code>-Lpath -ltvheap -rpath path</code>
	GCC	64	<code>-Lpath -ltvheap -Wl,-rpath,path</code>
IBM RS/6000 (all)	IBM/GCC	32/64	<code>-Lpath_mr -Lpath</code>
		32	<code>-Lpath_mr -Lpath --static_libKCC</code>
		64	<code>-Lpath_mr -Lpath</code>
AIX 4	IBM/KCC	32	<code>-Lpath_mr -Lpath path/aix_malloctype.so \</code> <code>-binitfni:aix_malloctype_init</code>
		64	<code>-Lpath_mr -Lpath path/aix_malloctype64_4.so \</code> <code>-binitfni:aix_malloctype_init</code>
	GCC	32	<code>-Lpath_mr -Lpath \</code> <code>path/aix_malloctype.so -Wl, -binitfni:aix_malloctype_init</code>
		64	<code>-Lpath_mr -Lpath \</code> <code>path/aix_malloctype64_4.so -Wl, -binitfni:aix_malloctype_init</code>
		32	<code>-Lpath_mr -Lpath path/aix_malloctype.o</code>
		64	<code>-Lpath_mr -Lpath path/aix_malloctype64_5.o</code>
Linux x86	GCC/Intel/PGI	32	<code>-Lpath -ltvheap -Wl,-rpath,path</code>
	KCC	32	<code>-Lpath -ltvheap -rpath path</code>
Linux x86-64	GCC/PGI	32	<code>-Lpath -ltvheap -Wl,-rpath,path</code>
		64	<code>-Lpath -ltvheap_64 -Wl,-rpath,path</code>
Linux IA64	GCC/Intel	64	<code>-Lpath -ltvheap -Wl,-rpath,path</code>
SGI	SGI/GCC/KCC	32	<code>-Lpath -ltvheap -rpath path</code>
		64	<code>-Lpath -ltvheap_64 -rpath path</code>
Sun	Sun/KCC/ Apogee	32	<code>-Lpath -ltvheap -R path</code>
		64	<code>-Lpath -ltvheap_64 -R path</code>
	GCC	32	<code>-Lpath -ltvheap -Wl,-R,path</code>
		64	<code>-Lpath -ltvheap_64 -Wl,-R,path</code>

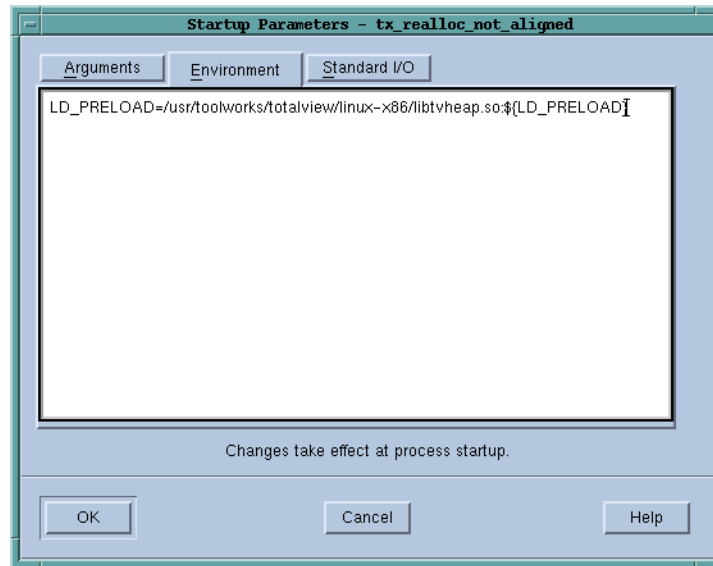
The following list describes the options in this table:

- path* The absolute path to the agent in the TotalView installation hierarchy. More precisely, this directory is:  
`installdir/toolworks/totalview.version/platform/lib`
- installdir* The installation base directory name.
- version* The TotalView version number.
- platform* The platform tag.
- path\_mr* The absolute path of the heap replacement library. This value is determined by the person who installs the TotalView malloc replacement library.



Since it is easy to misinterpret the path specifications, you may want to see what value TotalView uses when it sets a path. Here's the procedure:

- 1 Start TotalView.
- 2 Enable the Memory Debugger by selecting the **Tools > Memory Debugger** command, and then checking the **Enable memory debugging** checkbox.
- 3 Select the **Process > Startup Parameters** command and then select the **Environment** Page. Type a value that is the same as or similar to the one in the following figure:



## Attaching to Programs

When your program attaches to a process that is already running, the Memory debugger can not locate heap problems in that process unless you manually set a Memory Debugger environment variable. The variable that you use must be unique (or relatively so) on each platform. The following table lists these variables:

Platform	Variable
HP Tru64 Alpha	<code>_RLD_LIST</code>
IBM AIX	<code>MALLOCTYPE</code>
Linux IA64 and x86	<code>LD_PRELOAD</code>
SGI Irix	<code>_RLDN32_LIST</code> <code>_RLD64_LIST</code>
Sun	<code>LD_PRELOAD</code>

You can display the value that TotalView uses by displaying the **Environment** Page within the **Process > Startup Parameters** command. To set this variable:

- 1 Start TotalView and enable memory debugging.
- 2 Open this dialog box and see what the value is for your environment.
- 3 Close TotalView.

- 4 Start the program to which you will be attaching as an argument to the **env** command. For example, here's how to set this variable on AIX:

```
env MALLOCTYPE user:tvheap_mr.a totalview my_prog
```



*Do not set these environment variables so that the agent interposes itself when you execute any command. For example, use **env** to set this variable and run TotalView rather than **setenv**. If you use **setenv**, you will run the agent against all your programs, including system programs such as **ls**.*

## Using the Memory Debugger

This section describes using the Memory Debugger in various environments. This section describes the following environments and platforms:

- MPICH
- IBM PE
- SGI MPI
- RMS MPI

### MPICH

You use the Memory Debugger with MPICH MPI codes as follows. (Etnus has tested this only on Linux x86.)

- 1 You must link your parallel application with the Memory Debugger's agent, as described in "Linking Your Application With the Agent" on page 83. On most Linux x86 systems, type:

```
mpicc -g test.o -o test -Lpath -ltvheap -Wl,-rpath,path
```

- 2 Start TotalView using the **-tv** command-line option to the **mpirun** script in the usual way; for example:

```
mpirun -tv mpirun-args test args
```

TotalView starts up on the rank 0 process.

Because you linked in the Memory Debugger's agent, memory debugging is automatically selected in your rank 0 process.

- 3 If you need to, configure the Memory Debugger.
- 4 Run the rank 0 process.

### IBM PE

You can use the Memory Debugger with IBM PE MPI codes. There are two alternatives.



*You will not be able to install `tvheap_mr.a` under AIX on your target system unless you have installed the `bos.adt.syscalls` package, which is part of the System Calls Application Development Toolkit.*

The first is to place the following **proc** in your **.tvdrc** file:

```
# Automatically enable memory error notifications
# (without enabling memory debugging) for poe programs.
proc enable_mem {loaded_id} {
    set mem_prog poe
    set executable_name [TV::image get $loaded_id name]
    set file_component [file tail $executable_name]
```

```

        if {[string compare $file_component $mem_prog] == 0} {
            puts "Enabling Memory Debugger for $file_component"
            dheap -notify
        }
    }

# Append this proc to the TotalView image load callbacks
# so that it runs this macro automatically.
dlappend TV::image_load_callbacks enable_mem

```

Here's the second method:

- 1 You must prepare your parallel application to use the Memory Debugger's agent, as described in "Linking Your Application With the Agent" on page 83 and "Installing `tvheap_mr.a` on AIX" on page 88. Here is an example that usually works:

```

mpicc_r -g test.o -o test -Lpath_mr -Lpath \
    path/aix_malloctype.o

```

"Installing `tvheap_mr.a` on AIX" on page 88 contains additional information.

- 2 Start TotalView on `poe` in the usual way:

```

totalview poe -a test args

```



Because `tvheap_mr.a` is not in `poe`'s `LIBPATH`, enabling the Memory Debugger on the `poe` process causes problems because `poe` cannot locate the `tvheap_mr.a` malloc replacement library.

- 3 If you want TotalView to notify you when a heap error occurs in your application (and you probably do), use the CLI to turn on notification, as follows:
  - Open a CLI window by selecting the **Tools > Command Line** command from the Process Window showing `poe`.
  - In a CLI window, enter the `dheap -notify` command. This command turns on notification in the `poe` process. The MPI processes to which TotalView attaches inherit notification.
- 4 Run the `poe` process.

## SGI MPI

There are two ways to use the Memory Debugger on SGI MPI code. In most cases, all you need do is select the **Tools > Memory Debugging** command, select the `mpirun` process in the **Process Set** area, and then check the **Enable memory debugging** check box on the `mpirun` process. Occasionally, this can cause a problem. If it does, here's what you should do:

- 1 Link your parallel application with the Memory Debugger's agent, as described in the *Debugging Memory Problems* chapter of the *TotalView Users Guide*. Basically, the command you will enter is:

```

cc -n32 -g test.o -Lpath -ltvheap -rpath path \
    -lmpi -o test

```

- 2 Start TotalView on the `mpirun` process. For example:
 

```

totalview mpirun -a mpirun-args test args

```
- 3 If you need to, configure the Memory Debugger.
- 4 Run the `mpirun` process.

### RMS MPI

Here's how to use the Memory Debugger with Quadrics RMS MPI codes. (Etnus has tested this only on Linux x86.)

- 1 You do not need to link the application with the Memory Debugger because the **prun** process propagates environment variables to the rank processes. However, if you'd like to link the application with the Memory Debugger's agent, you can.
- 2 Start TotalView on **prun**; for example:  
`totalview prun -a prun-args test args`
- 3 Enable memory debugging by selecting the **Tools > Memory Debugging** command, selecting the **mpirun** process in the **Process Set** area, and then checking the **Enable memory debugging** check box. If you had linked in the agent, this option is automatically selected.
- 4 If you want TotalView to notify you when a heap error occurs in your application (and you probably do), check the **Stop execution when an allocation or deallocation error occurs** check box.
- 5 Run the **prun** process.

## Installing tvheap\_mr.a on AIX

---

You must install the **tvheap\_mr.a** library on each node upon which you will be running the Memory Debugger agent. One way to do this is to place a symbolic link in **/usr/lib** that points to the **tvheap\_mr.a** library. If you do this, you do not need to add special **-L** command-line options to your build. In addition, there are no special requirements when using **poe**.

The rest of this section describes what you need to do if you cannot create symbolic links. Even when you create symbolic links, you will still need to recreate **tvheap\_mr.a** whenever **libc.a** changes.

The **aix\_install\_tvheap\_mr.sh** script contains most of what you need to do. This script is in the following directory:

**toolworks/totalview.version/rs6000/lib/**

For example, after you become root, enter the following commands:

```
cd toolworks/totalview.6.3.0-0/rs6000/lib
mkdir /usr/local/tvheap_mr
./aix_install_tvheap_mr.sh ./tvheap_mr.tar /usr/local/tvheap_mr
```

Use **poe** to create **tvheap\_mr.a** on multiple nodes.

The pathname for the **tvheap\_mr.a** library must be the same on each node. This means that you cannot install this library on a shared file system. Instead, you must install it on a file system that is private to the node. For example, because **/usr/local** is usually only accessible from the node upon which it is installed, you might want to install it there.

The **tvheap\_mr.a** library depends heavily on the exact version of **libc.a** that is installed on a node. If **libc.a** changes, you must recreate **tvheap\_mr.a** by re-executing the **aix\_install\_tvheap\_mr.sh** script.

## LIBPATH and Linking

This section discusses compiling and linking your AIX programs. The following command adds *path\_mr* and *path* to your program's default LIBPATH:

```
xlc -Lpath_mr -Lpath -o a.out foo.o
```

When `malloc()` dynamically loads *tvheap\_mr.a*, it should find the library in *path\_mr*. When *tvheap\_mr.a* dynamically loads *tvheap.a*, it should find it in *path*.

The AIX linker allows you to relink executables. This means that you can make an already complete application ready for the Memory Debugger's agent; for example:

```
cc a.out -Lpath_mr -Lpath -o a.out.new
```

Here's an example that does not link in the heap replacement library. Instead, it allows you to dynamically set **MALLOCTYPE**:

```
xlc -q32 -g \
-L/usr/local/tvheap_mr \
-L/home/totalview/interposition/lib prog.o -o prog
```

The next example shows how you allow your program to access the Memory Debugger's agent by linking in the *aix\_malloctype.o* module:

```
xlc -q32 -g \
-L/usr/local/tvheap_mr \
-L/home/totalview/interposition/lib prog.o \
/home/totalview/interposition/lib/aix_malloctype.o \
-o prog
```

You can check that the paths made it into the executable by running the `dump` command; for example:

```
% dump -Xany -Hv tx_memdebug_hello
```

```
tx_memdebug_hello:
```

```
***Loader Section***
      Loader Header Information
VERSION#      #SYMtableENT      #RELOCent      LENidSTR
0x00000001    0x0000001f      0x00000040    0x000000d3

#IMPfilID      OFFidSTR      LENstrTBL      OFFstrTBL
0x00000005    0x00000608      0x00000080    0x000006db

***Import File Strings***
INDEX  PATH                                BASE                                MEMBER
0      /.../interpos/lib:/usr/.../lib:/usr/lib:/lib
1      libC.a                                shr.o
2      libC.a                                shr.o
3      libpthread.a                          shr_comm.o
4      libpthread.a                          shr_xpg5.o
```

Index 0 in the **Import File Strings** section shows that the search path the runtime loader uses when it dynamically loads a library. Some MPI systems propagate the preload library environment to the processes they will run;

others, do not. If they do not, you need to manually link them with the **tvheap** library.

In some circumstances, you might want to link your program instead of setting the **MALLOCTYPE** environment variable. If you set the **MALLOCTYPE** environment variable for your program and it fork/execs a program that is not linked with the agent, your program will terminate because it fails to find **malloc()**.

### Using the TVHEAP\_ARGS Variable

The values set within the **TV\_HEAP\_ARGS** environment variable allow you to control some of the Memory Tracker's behavior. Here are its arguments:

- backtrace\_depth** The maximum number of stack frames that the agent will record. The default is 32.
- backtrace\_trim** The number of frames to discard from the top of the stack. These frames are normally part of the Memory Tracker. The default is 5.
- display\_allocations\_on\_exit**  
Dumps a list of allocations that weren't freed when your program terminated.
- output {fd *fd-number* | file *pathname* }**  
Directs output to either a file descriptor or to a file. If you do not use this argument, the default is to send output to **fd 2**, which is **stderr**.
- verbosity *level*** Sets the verbosity level. If you do not use this argument, the default is 0 (zero). Here is what you can enter:
  - 0 No messages
  - 1 Writes *Starting* and *Finishing* messages. This lets you know that the agent is present.
  - 2 Writes event information as well as the breakpoint routine being called.

For example, you could set this variable as follows:

```
setenv TV_HEAP_ARGS "verbosity 2 output file foo.txt"
```







# Index

## Symbols

- <pending> pattern 48
- \_\_RLDN32\_LIST heap debugging environment variable 85
- \_RLD\_LIST heap debugging environment variable 85
- \_RLD64\_LIST heap debugging environment variable 85

## Numerics

- 0xa110ca7f allocation pattern 31
- 0xa110ca7f bit pattern 80
- 0xde110cf bit pattern 80
- 0xde110cf deallocation pattern 31

## A

- Add Filter dialog box 39
- adding and editing 39
- adding filters 38, 39
- Address not at start of
  - block problems 13
- agent's shared library 6
- aix\_install\_tvheap\_mr.sh script 88
- Allocate Paint Pattern dialog box 48
- allocated blocks
  - seeing 74
- allocation
  - 0xa110ca7f pattern 31
  - block painting 2
- allocation focus 40
- allocation location 19
- allocation pattern 80
- allocation point 47
- analyzing memory 60

- Apply pattern to allocations check box 49
- Apply pattern to deallocations check box 49
- Apply pattern to zero initialized allocations check box 49
- Apply Settings 51
- attaching to programs 85
- automatic variables 9

## B

- backtrace
  - deallocation 18
- backtrace ID 55
- backtrace option 74, 75
- Backtrace pane 53
- Backtrace View 55, 56
- backtrace\_depth TV\_HEAP\_ARGS value 82
- backtrace\_trim TV\_HEAP\_ARGS value 82
- backtraces 18, 25, 47, 53, 55, 75
  - depth 74
  - setting depth 75
  - setting trim 75
  - trim 74
  - which displayed 25
- bit painting
  - 0xa110ca7f 31
  - 0xde110cf 31
  - multiple precision 32
- bit pattern 47
  - in Variable Window 2
  - writing 2
- bit patterns
  - writing 26
- block information 47
- Block information area 18
- block length

- Graphical view 58
- block painting 2, 16, 26
  - defined 2
- Block Properties command 23
- Block Properties Window 23
- blocks, displaying 29
- breakpoints
  - internal 18
- bss data error 20

## C

- calloc() 47
- changing filter order 39
- Check interior pointers
  - during leak detection preference 54, 56, 60
- check\_interior option 79
- checking for problems 2
- CLI commands
  - dheap 63, 65
- columns
  - hiding 37
  - order 37
  - resizing 37
  - sorting 38
- commands
  - Process > Startup Parameters 85
  - Tools > Memory Event Details 18
- concealed allocation 14
- Configuration page 16, 27, 35, 36, 44
  - current settings tab 44
- criteria 40
  - adding to filter 39
  - backtrace entries 41
  - changing order 40

- exclusion 40
- matching 40
- operators 41
- property 41
- removing 39
- value 41

custody changes 15

## D

- dangling interior pointer 78
- dangling pointer problems 26
- dangling pointers 2, 78
  - example 27
- dangling pointers and leaks compared 12
- data section 5, 8
- data segment memory 61
- Data to Display preference 59
- deallocate
  - defined 8
- deallocation
  - 0xdea110cf pattern 31
  - block painting 2
- deallocation backtrace 18
- deallocation checkbox 24
- deallocation location 19
- deallocation pattern 80
- deallocation point 47
- deallocations, tracking 24
- depth, backtraces 75
- dheap
  - disable 63
  - enable 63
  - example 63
  - filter 63
  - hoard 63
  - info 63
  - leaks 63
  - nonotify 63
  - notify 63
  - paint 63
  - status of Memory Tracker 63
- dheap command 63, 65
- dheap –export command 77
  - data option 77
  - output option 77
  - set show\_backtraces option 77
  - set\_show\_code option 77
  - view option 77
- dheap –filter command 77
  - enable filtering 77
  - enabling a filter 77
- disabling all filters 39

- disabling while running error 45
- display\_allocations\_on\_exit TV\_HEAP\_ARGS value 82
- displaying block properties 23
- displaying blocks 29
- dont\_free\_on\_dealloc flag 73
- double free error 22
- dynamically allocate space 12

## E

- editing filters 38, 39
- Enable memory debugging 45
- Enable Memory Debugging check box 16
- enabling
  - Memory Tracker 17
- enabling all filters 39
- enabling filtering 37
- enabling filters 38
- enabling Memory Debugger 24
- enabling while running error 45
- environment variables
  - TV\_HEAP\_ARGS 82
- events, setting 45
- examining memory 28

## F

- fifo hoard queue 76
- filtering 38
  - enabling 37
- filtering heap information 77
- filtering, enabling 38
- filters 39
  - adding 38
  - adding criteria 39
  - allocation focus 40
  - backtrace entries 41
  - changing criteria order 40
  - criteria 39
  - criteria operators 41
  - criteria properties 41
  - criteria values 41
  - disabling all 39
  - editing 38
  - enabling all 39
  - exclusion 40
  - managing 38
  - matching criteria 40
  - naming 39
  - ordering 39
  - removing 38

- removing criteria 39
- sharing 39
- finding deallocation problems 13
- finding memory leaks 24
- flag
  - hoarded 73
- flags 73
  - dont\_free\_on\_dealloc 73
  - notify\_dealloc 73
  - notify\_realloc 73
  - paint\_on\_dealloc 73
- Fortran
  - tracking memory 6
- frames
  - eliminating 74
- free not allocated problems 13
- free problems 2, 74
  - finding 17
- freeing bss data error 20
- freeing data section memory error 20
- freeing freed memory 20
- freeing memory that is already freed error 20
- freeing stack memory error 20
- freeing the wrong address 21
- freeing the wrong address error 21
- freeing unallocated space 19

## G

- Generate View 36
- Generate View button 35
- Get Current Settings 50
- Graphical heap display
  - width in bytes preference 60
- Graphical View 29, 30
- Graphical view 57, 58

## H

- header section 5
- heap
  - defined 12
- heap API problems 74
- heap debugging 17
  - agent linking 83
  - attaching to programs 85
  - backtraces 70
  - enabling 17
  - enabling notification 73
  - environment variable 85

- freeing bss data 20
- freeing data section
  - memory error 20
- freeing memory that is already freed error 20
- freeing stack memory error 20
- freeing the wrong address 21
- freeing unallocated space 19
- functions tracked 17
- IBM PE 86
- incorporating agent 83
- interposition defined 5
- LIBPATH environment variable 84
- linker command-line options 84
- linking 15, 83
- linking the agent 83
- monitoring events 73
- MPICH 86
- preloading 6
- realloc problems 21
- RMS MPI 88
- setting environment variable 85
- SGI MPI 87
- starting 17
- stopping 17
- stopping on memory error 18
- tvheap\_mr.a library 88
- using 18
- heap displays, simplifying 77
- heap information filtering 77
- Heap Information page 58
- heap information, saving 77
- heap library functions 5
- heap memory 61
- Heap Status
  - Backtrace view 56
  - Graphical view 57, 58
  - Source view 56
- Heap Status Graphical View 29, 30
- Heap Status page 56
- Hide Backtrace Information button 23
- hiding columns 37
- hoard capacity 76
- Hoard Memory on deallocation check box 50
- hoard option 75

- hoarded flag 73
- hoarding 17, 26, 32, 49, 75
  - block maximum 76
  - defined 2
  - enabling 76
  - finding a multithreading problem 33
  - finding dangling pointer references 33
  - KB size 76
  - size of hoard 50
  - status 76

## I

- info option 74
- internal breakpoint 18
- interposition defined 5
- is\_dangling option 78

## L

- Label Leaked Memory preference 60
- LD\_PRELOAD heap debugging environment variable 85
- leak consolidation 79
- leak detection 79
  - checking interior 79
- Leak Detection page 12, 25
- leaks
  - concealed ownership 14
  - custody changes 15
  - defined 2
  - listing 2
  - orphaned ownership 14
  - underwritten destructors
    - leaks 15
    - why they occur 13
- leaks and dangling pointers compared 12
- leaks option 79
- LIBPATH and linking 89
- Library View
  - Memory Usage page 60
- line number 25
- linking 4
- linking the Memory Tracker agent 83
- linking with the Memory Debugger 15
- listing leaks 2
- Load 51
- load file 4
- Log all allocations on exit 50
- Log Memory Debug Information 50

## M

- machine code section 5
- MALLOCTYPE heap debugging environment variable 85, 89
- managing filters 38
- Maximum blocks to hoard field 50
- Maximum KB to hoard field 50
- memalign\_strict\_alignment\_even\_multiple\_TV\_HEAP\_ARGS value 82
- memory
  - analyzing 60
  - data segment 61
  - examining 28
  - heap 61
  - maps 3
  - pages 3
  - stack 61
  - text segment 61
  - total virtual memory 62
  - virtual stack 62
- memory block painting 16
- Memory Block Properties window 47
- Memory Blocks pane 52
- Memory Debugger
  - enabling 24
  - functions tracked 5
  - linking with 15
  - preferences 36
  - using 15
- Memory Debugging Command 5
- Memory Debugging Data Filters Dialog Box 39
- memory error notification 16
- Memory Event Details command 47
- Memory Event Details command. 19
- Memory Event Details Window 18, 24
  - Block information area 18
  - Point of Allocation tab 18
  - Point of Deallocation tab 18
- Memory Event Details window 46
- memory hoarding 17
- Memory Usage page 5, 60
- memory, reusing 75
- MPICH
  - and heap debugging 86

**N**

notification 15, 16, 18, 45, 74  
     disabling 73  
     enabling 73  
     not notifying 17  
 -notify option 74  
 Notify when deallocated  
     check box 24  
 notify\_dealloc flag 73  
 notify\_realloc flag 73

**O**

order of columns 37  
 orphaned ownership 14  
 output file for views 43  
 output TV\_HEAP\_ARGS  
     value 82

**P**

-paint option 79  
 paint\_on\_dealloc flag 73  
 painting 47, 79  
     allocation pattern 80  
     deallocation pattern 80  
     enabling 79  
     zero allocation 80  
 painting blocks 2  
 painting deallocated memory 33  
 pattern  
     <pending> 48  
 Pattern for allocations 47  
 Pattern for deallocations  
     field 49  
 PC, setting 13  
 Point of Allocation page 47  
 Point of Allocation tab 18  
 Point of Deallocation page 47  
 Point of Deallocation tab 18  
 pointers  
     dangling 2  
     passing 10  
     realloc problem 13  
 preferences 36  
     Heap Status  
         preferences 59  
 preloading Memory Debugger agent 6  
 Process > Startup Parameters command 85  
 Process Set area 25  
 Process Set selection 35  
 Process View  
     Memory Usage page 60  
 processes  
     limiting selection 36, 52

program  
     mapping to disk 3  
 programs  
     compiling 4

**R**

reachable blocks 79  
 realloc  
     pointer problem 13  
 realloc errors 21  
 realloc not allocated problems 13  
 realloc problems 21  
     finding 17  
 realloc() problems 13  
 reference counting 15  
 removing filters 38  
 reset backtrace hierarchy 55  
 resizing columns 37  
 Restart Enable button 17  
 restarting your program 17  
 reusing memory 75  
 running out of memory 14

**S**

Save 51  
 Save Configuration Page 50  
     Apply Settings 51  
     Get Current Settings 50  
     Load 51  
     Log all allocations on exit 50  
     Log Memory Debug Information 50  
     Save 51  
 saving heap information 77  
 saving view information 37  
 saving views 41  
 sections  
     data 5, 8  
     header 5  
     machine code 5  
     symbol table 5  
 selecting the process set 35  
 Set allocation focus level 53, 56  
 setting events 45  
 setting the PC 13  
 sharing filters 39  
 Show byte counts as megabytes (MB) or kilobytes (KB) preference 55, 56, 60  
 showing backtrace 74  
 showing backtraces 75  
 slave processes 73  
 sorting columns 38  
 Source View 52  
 Source view 56

Source/Backtrace page 59  
 space, dynamically allocating 12  
 stack frames 10  
     arranging 8  
 stack memory 11, 61  
 stack virtual memory 62  
 state information 74  
 -status option 73, 74  
 Stop execution when an allocation or deallocation error 17  
 Stop execution when an event or error occurs  
     check box 45  
 stopping when free problems occur 2  
 strdup allocating memory 13  
 symbol table section 5

**T**

-tag\_alloc 80  
 tagging 79, 80  
     notify on dealloc 80, 81  
     notify on realloc 80, 81  
 text segment memory 61  
 Tools > Block Properties command 23  
 Tools > Memory Debugging command 5  
 Tools > Memory Event Details command 18, 19, 47  
 Tools > Watchpoint command 26, 32  
 tracking deallocations 24  
 tracking memory problems 17  
 tracking realloc problems 21  
 trim, backtrace 75  
 TV\_HEAP\_ARGS environment variable 82  
     backtrace\_depth 82  
     backtrace\_trim 82  
     display\_allocations\_on\_exit 82  
     memalign\_strict\_alignment\_even\_multiple 82  
     output 82  
     verbosity 82  
 tvheap\_mr.a  
     aix\_install\_tvheap\_mr.sh script 88  
     and aix\_malloctype.o 89  
     creating using poe 88  
     dynamically loading 89

- libc.a requirements 88
- pathname require-  
ments 88
- relinking executables  
on AIX 89
- tvheap\_mr.a library 88

## U

- underwritten destructors  
15
- using the Memory Debug-  
ger 15

## V

- verbosity TV\_HEAP\_ARGS  
value 82
- version option 74
- View in Block Properties  
Window button 47
- views
  - output file 43
  - saving 41
  - saving backtraces 43
  - saving description in-  
formation 44
  - saving enabled filters  
43
  - saving information  
within 37
  - saving process infor-  
mation 43
  - saving source code in-  
formation 43
  - saving view descrip-  
tion 43
  - simplifying 77
- virtual memory 62
- virtual stack memory 62

## W

- watchpoints 27
- wrong address, freeing 21

## Z

- zero allocation 79
- zero allocation painting 79,  
80

